AD-A083 433 MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTE--ETC F/G 9/2
THE ARCHITECTURE OF AN OBJECT BASED PERSONAL COMPUTER.(U)
MAR 80 A W LUNIEWSKI N00014-75-C-0661
UNCLASSIFIED MIT/LCS/TR-232 NL

1 OF 3
AD A083433

LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-232

THE ARCHITECTURE OF AN OBJECT BASED PERSONAL COMPUTER

Allen William Luniewski

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under Contract No. N00014-75-C-0661

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

14 1. REPORT NUMBER
MIT/LCS/TR-232

2. GOVT ACCESSION NO.
AD-A083433

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
The Architecture of an Object Based Personal Computer

5. TYPE OF REPORT & PERIOD COVERED
Ph.D.Thesis-Dec.6, 1979

6. PERFORMING ORG. REPORT NUMBER
MIT/LCS/TR-232 ✓

10 7. AUTHOR(s)
Allen William/Luniewski

15 8. CONTRACT OR GRANT NUMBER(s)
N00014-75-C-0661

9. PERFORMING ORGANIZATION NAME AND ADDRESS
MIT/Laboratory for Computer Science ✓
545 Technology Square
Cambridge, MA 02141

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
ARPA/Department of Defense
1400 Wilson Boulevard
Arlington, VA 22209

11 12. REPORT DATE
Mar[illegible]980

13. NUMBER OF PAGES
267

14. MONITORING AGENCY NAME & ADDRESS(If different from Controlling Office)
ONR/Department of the Navy
Information Systems Program
Arlington, VA 22217

15. SECURITY CLASS. (of this report)
Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

This document has been approved for public release and sale; its distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
personal computer
capability addressing
structured programming
operating system
high level machine architecture

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This thesis proposes the architecture of a personal computer that provides better support than conventional architectures for recently developed concepts of structured programming. The architecture separates implementation and high level language issues. The architecture eliminates the need for an operating system by including, in a language independent manner, the features normally found in

4E7648

20.

operating systems. The architecture allows multiple languages to coexist safely. It is complete; the user has no need to leave the world defined by the architecture to solve a problem, including the important case of executing untrusted programs.

The architecture provides the semantic base needed by most languages. It supports a flexible execution environment that treats executable code and naming environments as objects. It explicitly supports the termination model of exception handling. A new mechanism, object viewers, provides type extension, access restriction and access revocation. The operating system features of process, inter-process communication/synchronization, permanent storage, I/O and system initialization/shutdown are provided.

An efficient implementation of the architecture is presented that is suitable for a personal computer. The implementation provides a large, real-time garbage collected object heap built out of a physical multi-level memory system.

Some ways in which the architecture can be used are shown. The focus is on showing how some problems of language implementation can be handled.

THE ARCHITECTURE OF AN OBJECT BASED PERSONAL COMPUTER

ALLEN WILLIAM LUNIEWSKI

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under Contract No. N00014-75-C-0661.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LABORATORY FOR COMPUTER SCIENCE

CAMBRIDGE MASSACHUSETTS 02139

The Architecture of an Object Based Personal Computer

by

ALLEN WILLIAM LUNIEWSKI

Submitted to the Department of Electrical Engineering and Computer Science on December 6, 1979 in partial fulfillment of the requirements for the Degree of Doctor of Philosophy

ABSTRACT

This thesis proposes the architecture of a personal computer that provides better support than conventional architectures for recently developed concepts of structured programming. The architecture separates implementation and high level language issues. The architecture eliminates the need for an operating system by including, in a language independent manner, the features normally found in operating systems. The architecture allows multiple languages to coexist safely. It is complete; the user has no need to leave the world defined by the architecture to solve a problem, including the important case of executing untrusted programs.

The architecture provides the semantic base needed by most languages. It supports a flexible execution environment that treats executable code and naming environments as objects. It explicitly supports the termination model of exception handling. A new mechanism, object viewers, provides type extension, access restriction and access revocation. The operating system features of process, inter-process communication/synchronization, permanent storage, I/O and system initialization/shutdown are provided.

An efficient implementation of the architecture is presented that is suitable for a personal computer. The implementation provides a large, real-time garbage collected object heap built out of a physical multi-level memory system.

Some ways in which the architecture can be used are shown. The focus is on showing how some problems of language implementation can be handled.

Thesis Supervisor: David D. Clark, Research Associate in the Department of Electrical Engineering and Computer Science

Keywords: personal computer, capability addressing, structured programming, operating system, high level machine architecture.

ACKNOWLEDGEMENTS

A five year PhD program is only completed with the aid and encouragement of a number of people. This is to acknowledge those who seem most important at this time. To those I have missed, I apologize and give my warm thanks.

First and foremost I must thank Dr. David Clark, my thesis supervisor, for his continual encouragement throughout my five years at M.I.T. especially the last two while I was working on this thesis. His many painful hours of reading and re-reading drafts of this document have resulted in a much better document than I alone could have produced. The remaining faults, of course, are my own. His help is gratefully acknowledged.

Professors Liskov and Svobodova, my thesis readers, have also contributed greatly to the production of a readable thesis. They have made many suggestions that have improved the quality of the presentation and the technical content of the thesis. Professor Svobodova'a meticulous reading of two drafts of this thesis have been of particular aid.

This thesis grew out of discussions held on Friday, January 13, 1978 between Warren Montgomery, Dave Reed, Karen Sollins and this author to design an object based computer. That group designed a machine known as the "510 Machine". The architecture presented in this thesis represents a significant evolutionary step from those initial discussions but still embodies much of the original philosophical approach. The help of this group of people is appreciated.

All of the members of CSR have, through the years, provided encouragement, enlightenment and the periodically necessary comic relief. In particular, I thank Steve Kent, Dave Reed and Karen Sollins who helped a rather hesitant first year graduate student become much more at ease in what was, at first, a new and unusual environment far from home.

I would also like to thank my family, especially my parents, who have been a source of strength throughout these five years. Their moral support, especially through the last three grueling months, has been cherished.

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under Contract No. N00014-75-C-0661.

CONTENTS

FIGURES

Chapter One

Introduction

In recent years there has been great interest in structured programming and in higher level languages that support the notions of structured programming. This thesis will describe a machine architecture that provides high level support for such languages, support much higher than the low level support provided by traditional Von Neumann computers. The thesis also will explore the inclusion of various operating system notions into such a high level architecture. The architecture will be designed for a personal computer both because such an environment is simpler than a time-sharing environment and because personal computers are likely to be the predominant form of computer resource in the future.

The major goal of the thesis, the design of a high level architecture, can be looked at in three ways. First there is the view that it is the architecture of an actual processor, e.g. the design of a piece of hardware that could be built. This viewpoint imposes both complexity constraints and completeness requirements on the architecture. The second view is that the architecture specifies an intermediate language that compilers for high level languages could use in compiling those languages. This viewpoint provides insights into the functionality that is needed in the architecture. A third, and unifying, view is that the architecture defines an interface that separates high level language issues (e.g. syntax issues, own variables and the degree of type checking provided by the language) from implementation issues (e.g. the representation of procedures, the control stack and managing the logical memory provided by the language). When viewed in this way, it is clear that the architecture should not contain language specific or implementation specific features as such features would reduce the architecture's utility as an interface specification.

1.1 Overview of the Thesis

The thesis will describe the design of a high level architecture for a personal computer. As motivated later in this chapter, the complexity of the personal computer must be limited due to economic constraints. The resources on a personal computer are noticeably finite; the user of the personal computer must frequently be aware of these bounds and be able to cope with them. The architecture must support the execution of multiple languages. Finally, the architecture must allow the user to run programs that he does not trust.

The architecture provides a garbage collected heap of objects of both built-in and user defined types, and provides facilities for flexibly constructing and executing programs. A built-in exception handling mechanism for the termination model of exceptions[30] is provided. The operating system notions of process, inter-process communication/synchronization, I/O, protection (including access revocation), storage quotas and system initialization/shutdown are supported in a consistent, language independent fashion.

The thesis also presents a sample implementation of the architecture; the implementation is based upon the work of Bishop[5] and Baker[3] for managing the logical memory provided by the architecture, and upon classical virtual memory techniques for mapping that logical memory onto a physical memory. Finally, the thesis provides examples of the utility of the architecture. First, though, the remainder of this chapter will motivate the need for the features described above and present concrete goals for the architecture.

1.2 The Personal Computing Environment

A current trend in the computer industry is the continuing decline in the cost of hardware due to the evolution of integrated circuits[4,46]. This trend indicates that the day when a computer can be given to every worker in an office (i.e. a personal computer) is approaching; the end of the shared, general purpose computer is near. Thus the architecture presented in the thesis will be designed with the needs of a personal computer in mind and not the needs of a shared computer. The personal computing environment has a profound effect on both the goals of the thesis and the ways in which those goals are met. This section presents the details and implications of the assumed personal computing environment.

1.2.1 Economic Considerations

A computer supplied to every person in an office must not cost a great deal; a computer that costs hundreds of thousands of dollars will never be a candidate for a personal computer. A realistic measure is that the computer must have a cost comparable to the cost of the terminal typically provided to users (e.g. on the order of a few thousand dollars).[1]

This economic constraint forces the architecture to be fairly simple; that is, the complexity of the hardware necessary to implement the architecture must be small. If the architecture is too complex, the implementation will be too expensive for a personal computer. However, the complexity constraint must not be taken to mean that the architecture must be as simple as current microprocessors. One of the purposes of the thesis is to set architectural sights somewhat higher and present an architecture that could be provided by hardware five to ten years in the future.

[1] These numbers must be considered only as estimates as the economic effect of eliminating a large central facility must be considered as must the factor of the economy of scale provided by such a facility. It is beyond the scope of this thesis to determine the break-even point. For this thesis it is only important to note, or perhaps hope, that a break-even point does exist and will be reached in the future.

1.2.2 The Use of Multiple Languages

The variety of languages that are to be executed on an architecture is an important design parameter of that architecture. An architecture designed to support only one language will differ from an architecture that supports many languages. Supporting only one language requires a great deal of confidence that future requirements will be met by that language. Moreover, the restriction to one language means that previously written programs can not be reused and makes it difficult to borrow programs since, in both cases, the programs may have been written in a language other than that chosen for the personal computer. This thesis takes the conservative approach of allowing multiple languages to be used on the personal computer.

The architecture will be best suited to supporting languages such as Alphard[52], CLU[31] and Euclid[26], which are strongly typed languages that support user defined types, since they seem to be typical of languages of the future. In addition, more traditional languages such as Algol and Pascal, which provide no general type extension facility, should be executable since their use is widespread. Finally, the architecture must consider the possibility that the user will write in assembly language.

There are two major implications of the assumption to support the execution of multiple languages. First, the architecture must have sufficient computational power to support the languages of interest. Simple computational completeness is not the issue, however. To permit reasonable language implementations, the architecture should provide the types and operations that are common to many languages.

Second, the key to supporting multiple languages is the amount of checking performed by the architecture. Possibilities for such checking are enforcing strongly typed variables, the "sealing" of extended type objects and access control checking. If the architecture only supported a single language, one that was capable of performing all needed checking at compile time, then the architecture would not need to perform any checking. On the other hand, attempting to perform all possible checks at run time is impossible in the case of multiple languages due to conflicting demands for checking. Also, providing too many checking features may so complicate the architecture that it is no longer economically viable as a personal computer.

Given these needs, there are two major ways to support multiple languages. First, a virtual machine architecture could be provided in which each language exists in its own, isolated virtual machine. Communication between languages would be by a built-in message passing mechanism that only permits communication using some built-in set of types (e.g. using ASCII text files to communicate between virtual machines). Such isolation does ensure the integrity of each language. Its one disadvantage appears to be the cost of transmitting large data structures between languages. This thesis takes the alternative approach of designing an architecture in which multiple languages can coexist. The architecture must supply checking facilities that permit one language to protect itself from other, untrusted, languages while, at the same time, being lax enough so that many languages are not excluded from using the architecture.

The architecture should provide both required and optional checking facilities for maximum flexibility. Since required checks are always performed, they must be compatible with all languages. The required checks must be sufficient to protect the integrity of the implementation of the architecture so as to preserve the architecture's utility as an interface specification (i.e. it must not be possible for malicious programs to cause the implementation to fail). These two requirements can be met by ensuring that all architecturally supplied operations are only invoked with valid operands. Such checking is sufficient to protect the integrity of the implementation but does constrain languages to live with these checks. Any other checks would further constrain languages but not aid in protecting the integrity of the architecture so they are not included in the architecture.

All other run-time checks are optional, to be performed when needed and otherwise avoided. Optional checks have two characteristics. First, the overhead associated with them must not be excessive when they are not being used. Second, the cost of using the facilities may be commensurate with their frequency of use; that is, if the checking is done infrequently, it can be space/time inefficient (perhaps with a very simple implementation) whereas if the checking is frequently performed, it should be space/time efficient (perhaps with a very clever implementation). The optional checks must be sufficient to allow one language to protect itself from the actions of other languages (e.g.

CLU must be able to protect CLU objects from being manipulated by Fortran programs except through the appropriate cluster interface).

1.2.3 The Issue of Trust - the User Versus Himself

One of the most striking features of the personal computing environment is the fact that it _is_ personal; the computer is "owned" by one person who is the only user of that machine. There is no sharing of resources (e.g. the computer) by mutually suspicious users (although see Kent[23] for an alternative viewpoint). As a result, the traditional protection and resource allocation functions of operating systems become much less important in this environment.

The actual allocation of resources by an operating system is relatively easy; attempting to insure some degree of fairness to the user community as a whole leads to complexity in operating systems. In a personal computer there is only one user, the owner of the personal computer, so that any request to allocate some resource is being made on behalf of that user. As a result, the allocation mechanism can be very simple in most cases since any allocation decision is fair.

In a shared computer it is important that protection constraints be provided by the system and enforced at all times. Failure to do so may lead to one user interfering, possibly in a malicious manner, with another user. In the personal computing environment such protection is unnecessary since there is only one user.

The previous paragraphs have implied that there is no need for architecturally provided resource allocation and protection mechanisms within a personal computer. However, this discussion has implicitly assumed that the user always trusts himself which is, unfortunately, not always the case. There are two cases of mistrust: while debugging programs and while running borrowed programs. In both cases there is a program running whose effect on the world is not known either due to possible incorrectness (the debugging case) or to lack of trust (a borrowed program). The importance of these two cases is unknown at this time since this kind of programming environment has not been available before. This thesis assumes that these will not be the normal case; rather, the user will usually be running programs that he trusts.

Although most programs are trusted, the architecture still must provide optional resource allocation and protection mechanisms that allow the user to protect himself against the potentially dangerous execution of untrusted programs. Due to the assumed infrequency of use of these mechanisms, they need not be extremely efficient in either space or time. However, their impact on the execution of trusted programs, the normal case, must be minimal.

1.3 The Nature of a High Level Architecture

The architecture presented in the thesis will be a high level one. This goal has two aspects. First, the architecture must meet the needs of the languages that are being developed today by supporting the notions of structured programming since that is the driving force behind most current language development efforts. Support for other, older languages is not a goal but, rather, is only desirable (i.e. attaining such support should not compromise the support for more modern languages).

Second, the thesis will explore partitioning the functions of an operating system into two classes. First, there are language/application independent features, which will be placed in the architecture itself. Second, there are language/application specific features, which will be built out of the features provided by the architecture. This partitioning allows traditional operating system features to be presented in a way that fits naturally into the rest of the architecture and in a way that is compatible with many languages and applications. Moreover, this partitioning will eliminate one part of the user's world, the operating system, leaving him with only the architecture, a language run-time system and an application.

These two issues are discussed in greater detail in the following sections. First, the implications of structured programming will be presented by discussing data and control abstractions as well as the issues of exception handling and storage semantics. Second, the implications of providing operating system features will be discussed. The issues of importance here are processes, inter-process communication/synchronization, permanent storage and resource allocation.

1.3.1 Structured Programming

Since the earliest days of computing the production of correct programs has been of major concern. Recently there has been increasing concern over the cost of software development and maintenance. These concerns have resulted in the programming philosophy known as structured programming.

Structured programming has frequently been referred to as "go-to less" programming[9] although it is better regarded as a philosophy of programming centered on the notion of abstraction[13]. An abstraction of an entity X is another entity Y whose properties are precisely those of X that are of interest to the user of Y. For instance, a television is a very complicated electronic device but the consumer sees a very simple abstraction of the actual device (on/off, channel selector, volume control, speaker and picture tube).

In the structured programming environment, the programmer first expresses problem solutions in terms of abstractions that closely model the application rather than in terms of the primitives of the programming environment. These high level abstractions are then implemented in terms of lower level abstractions. This process continues until the programmer has expressed the problem solution in terms of the primitives of the language being used. This method of program development, variously referred to as top-down design or stepwise refinement[6,8,51], is claimed to lead to programs that are easy to develop, show correct and maintain.

The languages associated with structured programming have two other important attributes: a well defined storage semantics and a built-in exception handling mechanism. The storage semantics of a language are of importance since they predefine the lifetime of objects and thus define the extent to which the user is responsible for managing the logical memory provided by the language.

Exceptions (or abnormal conditions) are to be expected during program execution since, occasionally, an unusual condition will arise (e.g. divide by zero, out of storage). Programmers must be aware that they may occur so as to produce correct programs. The inclusion of an exception handling mechanism in a high level language, and its use by the built-in features of the language, results in the user being aware of exceptions while writing his programs, thus leading to more correct programs.

The next four sections will discuss each of these issues of high level languages in turn. Each will motivate the need for some of the features of the architecture to be presented in chapter two.

1.3.1.1 Data Abstractions

A data abstraction is a collection of objects (data entities) and a set of operations on those objects, that models the application at hand so as to make the programming of that application easier. They are built out of previously defined data abstractions, including the primitive data types of the programming environment.

Data abstractions have been used for program construction since the earliest days of computers. The nature of data abstractions within programming languages has evolved since then resulting in their increased utility to aid in the production of correct programs. This utility means that any new computer system should support and encourage the use of data abstractions. This is done in three ways.

First, to support the use of data abstractions, the architecture must provide the programmer with the ability to create his own data abstractions. There should be no user visible distinction between the data abstractions provided as part of the architecture and those created by the programmer since any such distinctions will tend to give user defined abstractions second class status and thus discourage their use.

Second, the architecture must provide the means for objects of abstract type to be "sealed"; that is, provide a mechanism so that the underlying representation of objects may be manipulated only by the programs implementing objects of that type, that type's type manager. Such protection is important both for insuring the integrity of the type manager and for limiting errors caused by programs using objects of incorrect type. Although some languages provide this protection via compile time checking, sealing is a facility that is needed when multiple languages are present since all languages do not perform such checking.

Third, the architecture must encourage the programmer to use objects of abstract type so that the programmer can benefit from their advantages. By defining an object oriented world in which everything is an object, the architecture causes the programmer to think in terms of objects in order to use the architecture. Thus the use of objects will be "natural". The lack of distinction between built-in and user defined data abstractions, and the ability to easily construct new data abstractions, tends to enforce this pattern of thinking.

1.3.1.2 Control Abstractions

Structured programming has also considered the nature of the flow of control in programs. In fact, Dijkstra's concern in his original letter on structured programming[9] was control flow.

Through the years, the methods for specifying the flow of control within programs have evolved. The procedure has been constant through the years. It has changed in small ways, especially in regards parameter passing mechanisms, but the procedure has basically remained the same - a parameterized piece of code. There seems little need to change the procedural model for this thesis.

The means of controlling the flow of control within a procedure has, however, undergone a major philosophical change - the go to statement has been replaced with a rich collection of mechanisms to specify control flow in more structured ways. The effect of this philosophical change is that the code the user writes is linear since constructs such as while loops and if statements can be regarded as just single statements. This linear code results in a program that is easier to understand and write correctly.

The mechanism chosen to specify control flow must meet three criterion. First it must be complete. That is, it must be easy for the user of the architecture to represent the linear, conditional and iterative patterns of control flow needed to write programs. Second, the mechanism must be efficient. Since the mechanism will constantly be used, it is essential that it impose as little storage and execution time overhead as possible. Third, the mechanism should be restrained in its power by the lessons taught by structured programming. In particular, transfers of control should be regarded as actions local to a procedure and not as global, inter-procedural actions.

There are two possibilities for an intra-procedural control mechanism. One possibility is to derive a new control mechanism that is powerful, simple to use and efficiently implementable. Such a derivation is not, however, central to this thesis and so is rejected. The second choice, and the one taken in this thesis, is to use go to semantics. They are powerful, easy to use and can be efficiently implemented. The power of the go to will be tempered by only allowing transfers within the currently executing procedure, thus eliminating the most undesirable features of the go to, the non-local transfer of control.

1.3.1.3 Exceptions and Exception Handling

Procedure calls in early languages such as Algol and Fortran were always assumed to return with no error. Erroneous conditions were handled, if at all, by returning error codes to the caller. Since errors are a fact of life, it is essential that programmers be aware of them in writing programs. An exception handling mechanism that is supplied and used by the architecture encourages the user to be aware of exceptions in writing programs since the mechanism must be dealt with in order to use the architecture.

PL/1[22] introduced the notion that a procedure could signal a condition to inform a caller that something has gone wrong and allow the caller to fix the problem, if possible. After the handler for the condition finishes execution, the program raising the condition is allowed to continue execution. Since abnormal conditions do occur and the programmer should be aware of them, their introduction by PL/1 is an important contribution to the production of correct software.

CLU takes an alternative approach to exceptions by allowing a procedure call to terminate either normally or abnormally. A normal termination allows the caller to continue at the statement following the call. An abnormal termination indicates that the call has resulted in some unusual condition; the caller continues in an exception handler that is chosen based upon the name of the exception raised and a set of handlers associated with the abnormally returning call statement. The handler is responsible for evaluating the error and determining the appropriate action of the calling procedure.

There thus seem to be two models of exception handling in languages[12,28,30]: the termination model as in CLU and the continuation model as in PL1 or Mesa[36]. There does not seem to be any agreement in the language community as to the proper manner in which exceptions should be handled. For that reason the architecture should support both exception handling models. Moreover, the built-in operations of the architecture should raise exceptions so that the user is forced to be aware of them while programming.

1.3.1.4 Storage Semantics

The nature of the storage provided by a language is an important parameter of that language. Since the architecture presented in this thesis must be amenable to the various storage models of high level languages (recall that multiple languages are to be supported), it is important to review the various storage semantics of languages to be sure that the architecture supports the various storage models.

There are three classes of storage provided by languages: static, stack and heap. Static storage is storage that is associated with all invocations of a procedure. Every invocation of a procedure will have access to the same storage. The per-procedure storage in Fortran and Algol's <u>own</u> storage are examples of static storage.

Stack storage is data that is local to a particular invocation of a procedure. It is dynamically allocated when the procedure is called and destroyed when that invocation returns. Dynamic allocation is essential in languages with recursive procedures such as Algol. A stack allocation/deallocation strategy is frequently used to implement this type of storage, thus its name.

Finally, languages such as Lisp[33] and CLU have a heap oriented storage semantics. In these languages objects are created when needed and have a lifetime that is independent of the procedure that created them. The objects are deallocated only when all references to them have disappeared, this reclamation taking place through a process known as garbage collection[32].

The architecture should support all three types of storage. Of the three, the most general is heap storage. Given a heap oriented storage system, stack storage is provided by discarding all references to the dynamically allocated objects when the creating procedure terminates. Stack storage can thus be regarded as an implementation optimization of a particular pattern of use of a heap storage. Static storage is provided by arranging that all invocations of a procedure that has static data be given access to that data, which resides in the heap, whenever the procedure is invoked. As a result of its generality, a heap scheme will be used in the architecture presented in this thesis.

1.3.2 Operating System Issues

User programs typically do not run in the environment provided by the bare hardware. Rather, the hardware environment has been enhanced by the imposition of two layers of software, the operating system and a language run-time system, between the user and the hardware. The operating system serves three purposes. First, it serves to isolate the user from the inelegant aspects of the hardware such as I/O and managing the virtual memory. Second, it allocates resources to its users and does so in a way that attempts to be fair to all users. Third, it provides protection between users to protect one user from the erroneous or malicious programs of another user.

A language run-time system takes the environment provided by the operating system and adapts it to the particular needs of that language. In doing so, the run-time system may need to hide or drastically enhance and modify the semantics provided by the operating system. For instance, the operating system might provide block/wakeup as an interprocess synchronization mechanism while the language provides message passing semantics.

The thesis will explore eliminating the operating system and moving its functionality elsewhere. There are four reasons for doing so. First, the operating system frequently gets in the way of language run-time systems. Thus its elimination will simplify the job of these systems. Second, the features of operating systems that are moved into the architecture may be presented in a manner that is consistent with the way in which all other architecturally supplied entities are provided. Third, architecturally supplied features can be represented in a language independent fashion. Finally, it simplifies the users world by giving him two entities, the architecture and the language, to consider instead of three, these two and the operating system.

An operating system feature should be placed in the architecture if it is independent of both languages and applications. If a function is common to many languages, providing it in the architecture simplifies the implementation of those languages. When an operating system function involves a shared resource, e.g. a processor, languages can not be permitted to completely manage that resource due to possible errors in the implementation of those languages and due to the assumed mistrust of one language for another. Thus the management of shared resources must be architecturally provided. Finally, some features, e.g. I/O, must be provided by the architecture as no software is capable of providing it (i.e. the basis of some functions must be in the architecture even if in a primitive form). Languages and applications will provide the remaining functionality of operating systems by building upon the architecturally provided facilities.

The separation of function has the following effects. If the architecture is implemented as a piece of hardware, there should be little need for an operating system (and if it is needed it should be small and easy to implement). If the architecture is regarded as a compiler intermediate language, a certain degree of operating system independence for programs will result. Both occur because language independent operating system features will have been specified by the architecture; any mismatches between the hardware underlying the architecture and the architecture itself will have been handled by the implementation of the architecture and will not be of concern to the users of the architecture. The languages will be providing the language dependent features usually associated with an operating system.

1.3.2.1 Processes and Interprocess Communication

Operating systems provide processes (or tasks or jobs) that provide each user with one or more loci of control and, possibly, a separate address space in which each of the loci executes.

Processes are important to users for structuring problem solutions. They allow the specification of parallelism in the solution of a particular task and they allow the execution of multiple tasks in parallel. The first aids the user in solving the many tasks that have inherent parallelism and asynchrony in them. The second allows for more efficient use of resources. For instance, processes allow the user to perform a compilation in the background while interacting with a second process to edit a file. At the same time a third process could be listening to a communications network waiting for mail to arrive. The possibilities are numerous for the use of processes and allowing for them is important.

To be useful to the user, processes must be cheap. That is, the user should be able to utilize processes wherever they are natural with little performance penalty. This does not mean that the process model should be carried down to the level of very small activities as in Hewitt's actor model[17] since an efficient implementation of them is unknown; instead processes should be used for somewhat larger tasks. The correct size for processes seems to be somewhere between Hewitt's approach in which, conceptually, a process (called an actor in Hewitt's terminology) exists that adds two integers together and the Multics process[37], which is such a large entity that the user generally can only have one of them. Such a middle ground might be processes that correspond to the execution of a task such as an editor or compiler as in Unix[43].

Processes must be pro/ided at the architectural level in order to allow multiple languages to coexist. If languages provided all processes, there would be no way to guarantee that once one language began running, it would ever let other languages run. This is unacceptable in th. assumed environment where trust is not always present.

Parallel processes need to coordinate their activities (e.g. access to shared objects) as otherwise chaos results. The architecture must provide the basis for an inter-process synchronization mechanism as, ultimately, all inter-process synchronization mechanisms depend on some architecturally supplied facility, no matter how primitive.

1.3.2.2 Permanent Storage

In language systems, all of the storage associated with a program is destroyed when that program terminates. Long term storage is provided by the operating system and is outside of the scope of the language system. Since the user lives in a world in which long term storage is important (e.g. for inventory records or programs), it is important that the architecture provide the user with a notion of long term storage.

In traditional systems the user gets long term storage either via operating system guarantees (e.g. the programmer gives a file to the operating system and the operating system guarantees that the file will be there so long as the user desires it to be) or by gaining access to a raw storage medium (e.g. disk or magnetic tape). The first provides for permanent storage within the system while the second provides a means to transfer data outside of the system for safe keeping (or for simple transferal to another site). As both forms of permanent storage are important to users, the architecture must provide both.

The question is how to provide both forms of permanent storage. Due to the wide variety of I/O devices, the architecture must provide a means for the user to interact with each device individually to store and retrieve data. For system supplied permanent storage, it is sufficient for the architecture to treat any information given to it as permanent and keep it in as safe and permanent a manner as possible.[1]

1.3.2.3 Resource Allocation

A prime task of an operating system is to allocate the available resources to its users. Resources of potential interest include processor time, primary memory, permanent storage and peripheral devices.

Since processor and memory resources are provided by the architecture and since, even in the personal computing environment, their use needs to be controlled, the architecture must provide the means for controlling their allocation (no program or language can be entrusted with this task). The allocation of peripheral devices is not as important since users seldom use them. As a result, the architecture need only provide the means for resource allocation primitives for them to be constructed either by language systems or by applications. Again, due to mistrust, no program or language run-time system can be allowed to provide this basic function.

[1] The issue of backup and recovery is deliberately avoided here. It represents a research problem in and of itself and so is beyond the scope of this thesis.

1.4 Efficiency Considerations

If the architecture of the thesis is to be practical, it must be possible to efficiently implement it. An implementation need not run as fast as a large vector processor but it can not be as slow as a hand calculator either. A reasonable goal is that programs running on an implementation of the architecture run with a speed that is competitive with their execution time on more traditional architectures.

In judging the speed of an implementation, it must be remembered that this architecture is a higher level architecture when compared to present day processors. As a result, the comparison of average instruction execution times is not the right metric; the correct procedure is to compare instruction sequences that perform comparable tasks.

Another aspect of efficiency is memory utilization. The amount of physical memory required to perform some task while using the architecture should be comparable with the amount of memory required to perform the same task on a conventional computer. Inefficient use of memory comes from implementation overhead (information stored for the use of the implementation and not for the user) and from implementation inefficiencies (being unable to use all of the memory potentially available to store user data).

Both storage and processor inefficiencies are inevitable in providing the kind of environment proposed in this thesis. Overhead must, however, be minimized to make an implementation viable. Its impact, however, must only be judged in relation to the corresponding inefficiencies on conventional systems and not in an absolute manner.

1.5 Summary of Goals and Assumptions

This chapter has explored a number of issues related to the design of a high level architecture for a personal computer. This section will review the various assumptions and goals that have been presented.

The architecture of the thesis can be regarded as the specification of a structured programming machine. It must support the notions of structured programming that have been found most important: data abstractions and control abstractions. An exception handling mechanism is needed since abnormal conditions are a fact of life that programmers should be aware of. A heap storage will be provided by the architecture since it is the most general of the various possible storage semantics. Other storage semantics, such as stack and static storage, will be built using the basic heap mechanism.

The traditional operating system will disappear. Its functions (processes, inter-process communication/synchronization, permanent storage and resource allocation) will reappear in three places: the architecture itself, language run-time systems and in applications. The separation of traditional operating system function into language/application independent and dependent parts will be an important result of the thesis.

The architecture is being designed in the context of a personal computer. This means that the architecture must be economically feasible to build, thus limiting its potential complexity. This thesis assumes that multiple languages will be run on the personal computer resulting in the need for architecturally supplied checking to protect objects created by one language from unauthorized manipulation by programs written in another language.

Even in the personal computing environment, the user does not always trust himself or the programs he is running (e.g. debugging a program or running a borrowed program). As a result, the architecture must allow the user to execute such programs in a way so that they can only cause limited damage. This facility is assumed to be used infrequently so that its impact on normal operation must be limited.

An overriding assumption throughout this chapter, never explicitly stated, has been that of completeness. Basically, this thesis assumes that the user of the architecture only runs in the environment provided by the architecture; the user can not step outside of the architecture to fix a problem as he can in most language systems (where he can escape to an operating system provided interface). Thus the user must be able to prevent, detect and fix any and all problems that might arise while using the architecture itself. This implies the need for computational completeness as well as mechanisms for allowing the user to protect himself from himself. To a very great extent it is the completeness requirement that forces the various protection oriented facilities previously motivated to be provided.

1.6 Other Related Work

The previous sections have described the motivation, assumptions and goals of the thesis. In that process a variety of related work has been mentioned. This section will examine the relation between this thesis and other related work.

1.6.1 High Level Machine Architectures

This thesis is proposing the design of a high level architecture. A number of other researchers have investigated the area of higher level machine architectures.

McKeeman[34] has argued rather forcibly for machine architectures that are at a higher level than traditional von Neumann architectures. Others have proposed, and in some cases built, high level machines.

McMahan[35], in his PhD thesis, explored the design and implementation of an architecture that was oriented around Algol-68. Hoch[21] in his PhD thesis discussed an architecture that is designed to support the Gypsy language[1]. In 1973 a conference was held at the University of Maryland[38] to discuss high level language machine architectures. The architectures discussed in this conference were designed to primarily support one language with APL being particularly popular. All of the work described above has the property that the architecture is designed to primarily support a single language. The architecture of this thesis will be applicable to a wide variety of languages (although it would be improper to suggest that the architecture is not oriented towards a particular language, CLU in this case). This thesis also differs from the cited work in that it addresses a wider variety of issues than just language oriented ones - in particular, the various operating system issues mentioned earlier are of importance.

Bishop[5] and Snyder[47] have designed systems for supporting an object oriented style of programming. In both cases, however, the principle concern has been the solution of the memory management problem - the management of a large heap of objects that reside both in primary and secondary memory. In both cases, the architecture is just a framework for posing the memory management problem. In this thesis, the principle concern is the architecture itself; the implementation concerns are of a secondary nature.

1.6.2 Capability Machines

This thesis also is related to the various work that has been done on capabilities and capability based machines. Capabilities were originally proposed by Dennis and Van Horn[7] as a protection mechanism and are nothing more than unforgeable pointers to objects (segments in the cited paper). Fabry[11] investigated the use of capabilities as an addressing mechanism.

There have been (at least) three capability based systems reported in the literature. The Cal-TSS system[25] was a capability based system that ran on top of a CDC-6600. All of the capability mechanism was provided by software. The CAP system[50] runs on a special purpose processor that implements capabilities in hardware. Hydra[53] is an operating system that runs on a multiprocessor configuration of PDP-11's that provide minimal support for capabilities in the form of address mapping hardware. All three of these systems have taken a conventional architecture and added capabilities to that architecture for the purpose of providing protection facilities. For the most part, user programs see a conventional execution environment. This thesis differs from these machines in that capabilities form the basic addressing architecture at all times during the execution of user programs. Moreover, the approach of this thesis has roots both in language considerations and in operating system issues.

1.7 Thesis Outline

The remainder of the thesis presents an architecture that meets the goals presented in this chapter. The nature of the thesis is such that a great deal of material must be presented in order to demonstrate that the goals of the thesis have been met. The reader is asked to be patient until the end when the entire thesis can be placed in perspective.

Chapter two will present the architecture itself. Chapters three and four will demonstrate one possible implementation of the architecture. Chapter three will concentrate on the logical memory management issues involved in implementing the architecture. These are the hardest implementation problems as they require the creation of algorithms, rather than simply data structures. The algorithms presented are based upon the work of Bishop[5] on garbage collecting large address spaces and the real time garbage collector of Baker[3]. Chapter four will concentrate on the remaining issues involved in the implementation: management of physical resources, especially memory, implementing the basic types of the architecture and some possible hardware assists to produce an efficient implementation. Chapter five will show some ways in which the architecture can be used. The focus is demonstrating the unusual aspects of the architecture, especially in regards to the implementation of languages on the architecture. Chapter six will review the results of the thesis and attempt to show why the goals of the thesis have been met. It will also propose some areas for further research.

Chapter Two

The Architecture

Chapter one has presented the primary goal of the thesis - the design of a high level machine architecture. This chapter presents an architecture that conforms to this goal.

A major goal is to provide support for objects at all levels. In fact, the goal is to present users with an object oriented view of the world that includes both large objects, such as files as in CAP and Hydra, but most importantly very small objects, such as integers as in CLU. For this reason, this architecture can be characterized as a small object processor and thus will be named AESOP (An Experimental Small Object Processor).

In AESOP, everything will be an object. This includes not only integers, booleans and vectors, but also unexpected things such as processes and procedures. Even operations that control execution (e.g. procedure calls and go-to statements) will be provided as operations on some object. Thus AESOP is defined by the types it provides and the operations provided on objects of those types.

The set of proposed basic types and operations is intended to be representative of the types that the architecture should, and in some cases must, support and not a definitive set. The computational data types (e.g. booleans) may be modified or replaced so long as sufficient computational power remains to meet the needs of users. The remaining types are essential as each is representing some part of a complete system as a type. However, the reader should feel free to augment or modify this set so long as the various parts of a computer system continue to be represented.

In discussing the basic types, two liberties are taken for expository simplicity. First, not all operations on a given type are presented in this chapter or in one place. Instead, only the operations essential to the current discussion are given. Appendix A should be consulted for a complete list. Second, the description of operations ignores the possibility of an operation returning abnormally (e.g. attempt to divide by zero). All operations detect and signal, via a built-in exception handling mechanism, all of the errors that the reader would expect. Again, Appendix A should be consulted for a complete listing.

The following syntax will be used to present operations:

type_name$operation (input parameters) returns (results)

For example, A$B(W, X) returns(Y, Z) defines the B operation on objects of type A to take two input values (W and X) and return two values (Y and Z). Each parameter (result) may have a type specification to indicate an expected (returned) type (e.g. X:integer specifies X as an integer). If no such specification exists, the type of the argument (result) is not constrained by the architecture.

This chapter is divided into two distinct parts. The first three sections discuss basic AESOP semantics. The remaining sections discuss higher level notions that are needed to meet the completeness goals of chapter one. Section one discusses naming in AESOP - what it means to say that one object contains a reference to a second object. Section two presents AESOP's computational data types. Section three discusses execution in AESOP: what is executable and how it executes. Section four presents a mechanism that uniformly treats the creation of objects of extended type, control of access to objects and the revocation of access to objects. Section five discusses those features of AESOP that are subsuming traditional operating systems: processes, interprocess synchronization, storage management, I/O, system initialization and system shutdown.

2.1 The Naming Architecture

A fundamental architectural decision is how objects are named by other objects. The basic naming mechanism will define the extent to which two objects can share access to a third obiect and will define the ability of programs to acquire the names of objects.

AESOP allows an object to directly refer to any other objects within AESOP by containing the names of those objects. A name is an unforgeable pointer to an object (i.e. names are like capabilities). Unforgeable names have been chosen since they limit errors by preventing the construction of arbitrary pointers and provide a limited form of protection since a program can not access an object unless it has been given the name of that object. Unforgeable pointers are one tool used in AESOP to aid the user in not injuring himself.

Note that objects contain the names of objects and not the values of objects. As a result, two objects may share access to a third object. This permits the construction of arbitrary graphs of objects, including the important case of cyclic data structures such as circular lists. Chapter four shows how objects may efficiently refer to certain built-in objects (e.g. integers) so that the indirection implied by this naming scheme is not an efficiency problem.

2.2 The Basic Computational Data Types

To support effective computation, AESOP must provide some data types to compute with and a data aggregation facility to allow the creation of complex data structures. This section describes a set of types for AESOP that meets these needs.

For computational purposes, AESOP will provide booleans, characters (which are not character strings) and a possibly finite subset of the mathematical integers. All of the expected operations will be provided. In addition, the null type, which has one object (nil), exists. The only operation on nil is to ask if it is nil so a reference to nil is, in essence, a reference to nothing. It is useful for cyclic data structures (e.g. for marking a leaf node of a tree).

A vector, AESOP's data structuring facility, is used to aggregate a number of objects into a larger one. It is an ordered sequence of names, referring to objects of potentially arbitrary type that is indexed by positive integers. Thus a character string would be formed by creating a vector of some length and then filling in the entries in the vector with the names of characters.

Some operations on vectors are presented in Figure 2.1. There are two aspects of vectors that are of note. First, an initial value must be supplied whenever a vector is created or its length increased (by a modify operation) so that all elements of a vector are always initialized. This permits an AESOP implementation to be more space and time efficient since it need not be concerned with uninitialized vector entries.

Second, vectors may be restricted to objects of one type through the nature and new_nature arguments. If these are nil, v may refer to objects of arbitrary type. Otherwise nature refers to a type manager, the AESOP object that implements objects of a given type (see section

```
vector$create(size:integer, nature:tm, initial_value) returns(v:vector)

vector$new_status(v:vector, new_nature:tm, new_size:integer,
   initial_value)

vector$ref(v:vector, i:integer) returns(value)

vector$status(v:vector) returns(size:integer, nature:tm)

vector$store(v:vector, i:integer, value)
```

Figure 2.1. Some of the operations on vectors.

2.3.2.3), and all elements of v are restricted to refer to objects of that type. The ability to restrict the types of vector entries is included for two reasons. First, it permits the architecture to perform some checking for the programs running on AESOP. Second, it permits the implementation to optimize the storage of certain vectors (e.g. a vector of booleans) as will be seen in chapter four when the implementation of vectors is discussed.

2.3 Execution in AESOP

The basic executable unit in AESOP is called an execution triple for reasons that will be clear shortly. This section will answer three questions: What can be executed? How does execution begin? Once begun, how is control flow within an execution triple handled?

The discussion has three distinct parts. First, the fundamental components of execution are discussed by describing the nature of code and the way in which executing code refers to objects. Second, the ability of the user to treat these components as objects, to create an execution triple from such objects and to cause the execution triple to begin execution is discussed. Third, the means by which control flow within an execution triple is controlled is discussed including how an execution triple ceases execution, either normally or abnormally, and returns values to its invoker.

2.3.1 The Execution Triple

An execution triple is the basic executable unit of AESOP. It is not an AESOP object but is, rather, an object that is conceptually hidden by AESOP and simply used to express execution in AESOP. Conceptually, an execution triple is presented to the code interpreter of a process for execution so as to change the state of that process. Thus the process type manager is responsible for executing execution triples.

An execution triple consists of three components. First is a code segment, an object that contains the actual instructions executed by an AESOP processor. The second and third components specify the naming environment in which that code will execute.

2.3.1.1 Code Segments

Objects of type code segment contain the instructions that are executed in an execution triple. Code segments are created via the operation:

code_seg$create(rep) returns(cs:code_seg)

where rep describes the contents of the code segment. The form of the input to create is discussed later in this chapter. Create is the only operation on code segments, so code segments are immutable, and the errors self-modifying code can cause are avoided.

A code segment is an ordered sequence of instructions each of which is a call on some type manager. In AESOP, an object can only be manipulated by its type manager and not by any program that decides it should examine the object. Thus it makes no sense to have an instruction that is not a call on a type manager as it could not do anything. At first the reader might object that an operation such as procedure call is needed. However a procedure is just an object and, as such, operations on it are provided by its type manager. So, in fact, the only possible instruction is a call type manager instruction.

2.3.1.2 The Naming Environment for Execution Triples

Since a code segment is an object, it may contain the names of other objects. Thus objects may be referred to directly during execution. In addition, a code segment may refer indirectly to objects by using the other two components of the execution triple, the local name space (LNS) and the global name space (GNS).[1] Such a reference is an ordered pair of the form (name-space, index) where name-space specifies either the current LNS or the current GNS and index is a small integer used to select the index'th name from the indicated name space. Figure 2.2 shows an example of an execution triple P that is executing in an environment with the given code, LNS and GNS. By executing using a different LNS and/or GNS, the indirect object references in a code segment change meaning (i.e. they are context dependent names with the

[1] Name spaces are similar to Hydra C-lists[54] and CAP indirectories[50].

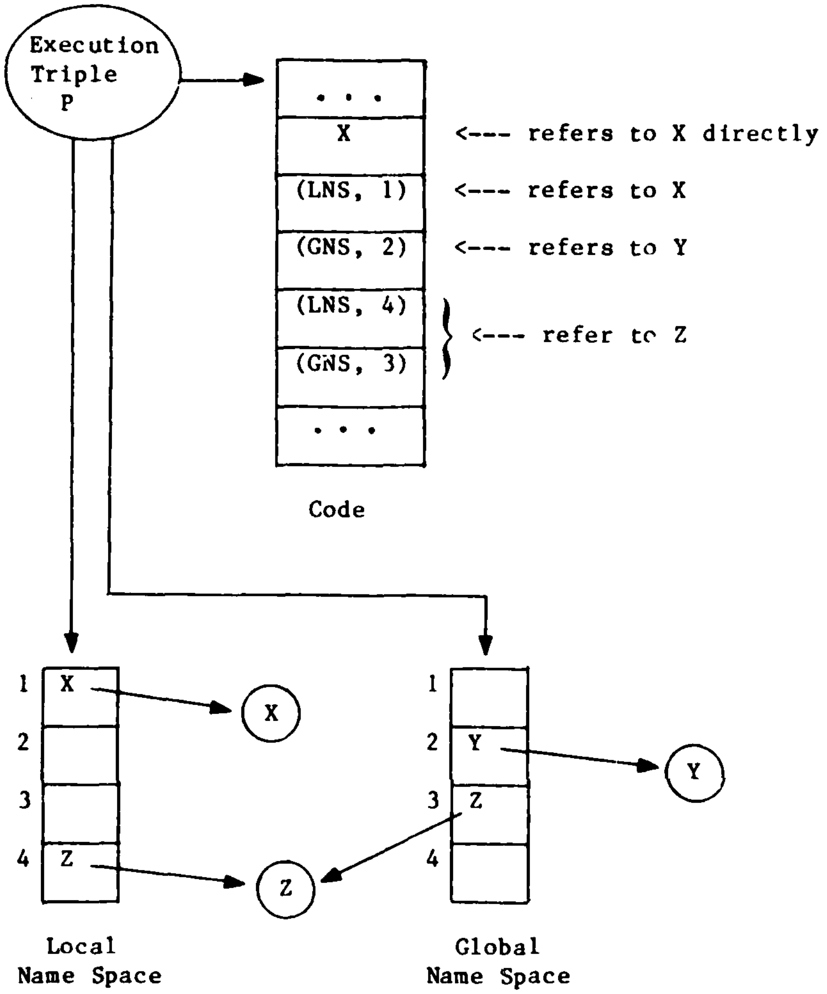


Figure 2.2. An example of the use of name spaces.

LNS and GNS as the context) while the direct references are unchanged (i.e. they are context independent).

A name space is nothing more than a vector of names of objects of arbitrary type. A name space is distinguished from other vectors only by its use in an execution triple; it may still be manipulated as a vector. The ability to treat vectors as name spaces and conversely is very useful as will be seen in the next section.

The importance of name spaces becomes most clear when procedures are considered. As will be seen later in this section, invocation of a procedure causes execution to continue in a new execution triple that has a newly created LNS in it. The per-invocation LNS means that procedures are reentrant since the LNS may contain the names of objects that are local to the current invocation of the procedure (e.g. local variables and parameters) by reserving an entry in the LNS for each of the procedure's local variables and parameters.

The GNS will contain references to objects that need to be known to most procedures within a process (e.g. operating system interfaces and language run-time support routines). This use of the GNS represents an important space optimization - "factoring" common object references out of the LNS's and code segments of a collection of procedures. More importantly, the resulting GNS is a virtual machine interface for those procedures since they expect certain GNS entries to contain references to objects with certain properties (e.g. entry 3 might be a log routine). By creating a name space containing the names of a new set of support routines and running a procedure with that name space as a GNS, it is possible to present a new virtual machine interface to that program. This feature is useful for debugging programs and for encapsulating programs. For instance, while debugging a new database manager, a user might want to give that program access to the file system through a program that filters requests to the file system for validity and generates an audit trail for debugging purposes. By running the database manager in a GNS that contains the name of the filtering file system interface, and not the regular file system interface, the filtering and auditing is accomplished in a simple manner that is transparent to the database manager. Thus the GNS is another means for the user to protect himself from himself.

2.3.2 The Creation of Execution Triples

The creation of an execution triple requires the specification of a code segment and two vectors, one to act as an LNS and one to act as a GNS. AESOP provides three means of creating execution triples: by explicitly specifying all three components, by procedures and by type managers.

2.3.2.1 Closures and Components as Objects

Closures are the first means of creating an execution triple. A closure is a triple consisting of a code segment and two vectors, one an LNS and one a GNS. Closures are, in essence, pieces of code that come with a fixed, unchanging naming environment. The operations on closures are given in Figure 2.3. The create operation makes a new closure out of its arguments. The run operation causes closure cl to be executed

```
closure$create(cs:code_segment, lns:vector, gns:vector)
   returns(cl:closure)

closure$run(cl:closure) returns(res(1), ..., res(N))
```

Figure 2.3. The operations on closures.

with its component objects forming the execution triple. Conceptually, the closure is unpacked and an execution triple created and passed to the process code interpreter for execution. Any results returned by cl are placed in the res as described later in this section.

Closures provide two facilities. First, they provide protection when an executable body of code needs to be passed through an untrusted intermediary since closures are inviolate. Second, programs can create vectors and code segments of any sort and pass them to closure$create to create a closure that can subsequently be executed. The resulting execution triple can be of arbitrary form, thus giving programs complete control over the contents of execution triples. This can be used both to create execution triples for immediate execution and to permanently bind a code segment to an execution environment for subsequent execution.

2.3.2.2 Procedures

Procedures provide procedural abstractions (i.e. parameterized pieces of code) and are the second way to create execution triples. From the point of view of execution, procedures provide a simple way of creating a particular stylized execution triple, one with a per-invocation LNS to handle parameters and local variables.

A procedure is a pair of elements: a template LNS and a code segment. When the procedure is called, the template LNS is copied into a newly created vector and the copy becomes the LNS for the procedure activation. (Note that this allows procedures to be reentrant.) The code segment in the procedure's representation is the code segment that will be executed when the procedure is called and is the body of the procedure. The operations on procedures are given in Figure 2.4.

The create operation creates a new procedure, p, with cs as its body and t_lns as its template LNS. P requires at least min_args arguments but no more than max_args arguments (if max_args is nil, an arbitrary number of arguments are permitted). Start specifies the location in p's LNS where its parameters are to be placed when it is invoked.

The call operation invokes p, passing the arguments {arg(i)} as parameters to p, by the following scheme. First, an LNS, call it L, for the invocation of p is created by copying p's template LNS. Next, L[1] is set to L to allow the procedure to refer to its own LNS. Next, L[start] is set to the actual number of arguments passed. The arguments are passed to p by setting L[start+i] to arg(i). An execution triple consisting of code segment cs, L as its LNS and the GNS of p's caller as its GNS now begins execution. Any return values are placed in locations in the caller's naming environment as specified by {res(i)}. When P

```
proc$call(p:proc, arg(1), ..., arg(N)) returns(res(1), ..., res(M))

proc$call_with_gns(gns:vector, p:proc, arg(1), ..., arg(N))
   returns(res(1), ..., res(M))

proc$create(cs:code_segment, t_lns:vector, min_args:integer,
   max_args:integer, start:integer) returns(p:procedure)
```

Figure 2.4. Some of the operations on procedures.

returns, either normally or abnormally, L is destroyed, invalidating all outstanding references to it. The destruction of L permits an AESOP implementation to handle the LNS's associated with procedure invocation in a stack manner for efficiency. If N is not within the range of expected number of arguments to P, an exception is raised and P not invoked. If P returns normally with R results, only the first min(M, R) res(I) are assigned to.

The call_with_gns operation calls p as for the call operation except that the procedure runs with gns as its GNS. This allows p to be executed using a specially constructed GNS for debugging or encapsulation purposes.

2.3.2.3 Type Managers

A type manager is a set of procedures, one for each operation that the type manager supports, that defines a type. AESOP provides type managers to permit the construction of extended types, types not built into AESOP. When an attempt is made to use an operation on a type, the procedure that implements that operation is extracted from the set and called to perform the operation. Type managers are thus a third means for specifying an execution triple, this time by indirectly specifying a procedure.

The important operation on type managers is the create operation:

```
   tm$create(proc(1):proc, ..., proc(N):proc) returns(t:tm)
```

where tm is an abbreviation for type_manager. It creates a new type manager, and thus a new type, that has proc(i) as the procedure implementing its i'th operation. Operations provided by type managers are invoked by AESOP instructions (recall that every instruction invokes a type manager).

2.3.3 Control Flow

The previous section showed how new execution triples are submitted for execution. When an execution triple begins execution, the first instruction of its code segment is executed. This section discusses the subsequent flow of control, i.e. which instruction is executed next and how an execution triple ceases execution and returns values to its caller.

2.3.3.1 Returns from Execution Triples

An execution triple can return to its caller in one of two ways: normally or abnormally. A normal return is effected either by executing the operation process$return(<results>)[1] or by falling off of the end of the code segment. In either case, execution of the caller resumes at the instruction immediately after the call. The first may return results to the caller while the second does not. If the number of returned results (say R) does not match the number of results that the invoker expected (say M), only min(M, R) results are actually returned.

[1] Recall that all operations concerning execution triples are provided by the process type manager. The process type manager is discussed further in Section 2.5.1.

The invoker is responsible for detecting that a mismatch in the number of returned arguments has occurred.

An abnormal return (i.e. "raising" an exception) is effected by the operation process$signal(signal, argument). It causes the caller to be returned to at its abnormal return point. The argument signal is returned as the signal name and argument is returned as the argument to the handler. Only a single argument is provided to simplify the architecture; more complicated signal arguments must be provided for by "packaging" multiple values into a vector and returning that vector. Exception handling is discussed in more detail after discussing intra-procedural control flow.

2.3.3.2 Intra-procedural Flow Control

The basic unit of execution in AESOP is the code segment, a linearly ordered sequence of instructions. As a code segment is being executed, the normal course of action is to execute the instructions in order, starting with the first one. There is, however, a need for non-sequential execution to handle the conditional execution of code.

As motivated in chapter one, AESOP uses a restricted form of "go-to" semantics in which transfers are only permitted to another instruction within the same code segment. Transfer of control within a code segment occurs in either a conditional or unconditional fashion. Unconditional transfers are handled by the operation:

```
          process$transfer(offset:integer)
```

which causes execution of the current code segment to continue at the instruction that is "offset" instructions (positive or negative offset) from the current instruction. Thus process$transfer(+2) skips execution of the next instruction while process$transfer(0) is a null, infinite loop. No explicit process or code segment argument is given so that only execution of the current code segment, in the current process, is effected. This restriction eliminates the most dangerous aspect of go-to semantics, the non-local go-to.

Conditional transfers are also provided by the process type manager. For example, the operation:

process$boolean_branch(b:boolean, if_true:integer, if_false:integer)

branches using either if_true or if_false as the offset depending on the value of b. Other, similar operations, can be imagined but will not be discussed here - see App^endix A for a complete proposed list.

2.3.3.3 Exceptions and Exception Handling

Chapter one motivated the need for AESOP to provide the means for handling exceptional conditions during execution. There are two models of exception handling: the continuation model (as in Mesa[36] for instance) in which the program raising the exception continues after the handler for that exception terminates and the termination model (as in CLU for instance) in which the program raising the condition terminates by the act of raising the exception.

In the continuation model, exception handlers are little more than procedures. This model can be achieved by passing along with every call a procedure that will handle any exceptions that are raised. The act of raising an exception becomes the calling of the procedure that was passed as an argument. Since it is so easy to obtain, the continuation model is dismissed at this point from an architectural point of view.

The termination model can, however, use some architectural help since the only means to implement it within the mechanisms thus far presented is to pass error codes as return arguments and check them on return (i.e. call P(..., code); if code ^= 0 then <error handler>). This is a workable, but inelegant, scheme. It is also probably not as space or time efficient as a scheme built into the architecture. So instead, associated with every instruction will be the offset of the instruction that should be transferred to (via a forced "go-to" operation) whenever the called procedure makes an abnormal return. For instance, the "instruction":

```
integer$add(a,b,c) except(+10) normal(+1)
```

causes execution to continue at the next instruction whenever integer$add returns normally but 10 instructions following this instruction if integer$add returns abnormally. Thus the instruction 10 instructions after this one is the initial instruction of the exception handler. Once the architecture does the forced transfer to the handler, it forgets that the exception and subsequent transfer has occurred; it acts as if the program itself had executed the transfer instruction.

When an execution triple returns abnormally (via the previously described process$signal operation), the arguments to that operation are placed in entries 2 and 3 of the LNS of the program that invoked the execution triple. This information allows the handler to determine what the error was and act accordingly.

Note that if the exception handler should invoke a procedure that returns abnormally, the information passed to the initial handler will be overwritten by the information passed to the new handler. An automatic stack of exception handler information is not provided as it introduces a great deal of implementation complexity to handle an event that can be avoided by proper programming. Since most code segments will be provided by compilers, the compilers can handle this problem in a manner that is transparent to programmers.

2.3.4 The AESOP Instruction Set and Code Segments

Now that all of the parts of the AESOP execution environment have been presented, the nature of AESOP instructions can be given. Basically, an instruction must specify a type manager to be called, an operation to be performed, arguments to be passed, where results are to be placed, a normal return point and an abnormal return point. This section will discuss the information that must be supplied, in the rep object, for each instruction in the code segment that code_seg$create will create so as to permit subsequent execution of the code segment in accord with the mechanisms presented earlier in this section.

The basic format of an instruction is given in Figure 2.5. The effect of the instruction is to call some type manager, passing the

Type Manager Spec.	Operation Number Spec.	Normal Return Point	Abnormal Return Point	Argument Spec. (≥ 0)	Result Spec. (≥ 0)

Complete set of operands to an instruction.

Name	"LNS"	Index	"GNS"	Index	a literal

The forms for all fields except results.

"LNS"	Index	"GNS"	Index

The forms for a results specification.

Figure 2.5. Complete options for instructions.

objects specified in the instruction as arguments and possibly returning some objects as results.

The name of the type manager to be called is specified by the type manager field. The first form permits explicit reference to the type manager to be included in the instruction. The second form retrieves the index'th entry from the current LNS, which must name a type manager, and invokes that type manager. The third form works similarly by retrieving a reference to a type manager from the current GNS. The fourth form allows the specification of a literal (e.g. a character string) that names a built-in type manager. In this case, code_seg$create interprets that literal to determine which type manager to invoke and, conceptually, places a reference to that type manager in the produced code segment.

The operation number, normal return point, abnormal return point and arguments are specified in a similar manner. In these cases, however, the literal may denote any built-in object (e.g. the character string "7" may denote the integer 7). The presence of literals not only permits easy access to built-in objects but also allows an implementation of AESOP to optimize instructions based upon known values passed to code_seg$create.

The operation number field specifies which operation is to be performed, by giving the positive integer, call it i, that is the "name" of the operation. The i'th procedure passed to the tm$create operation that created the invoked type manager will be invoked.

The normal return point specifies, as an offset relative to the current instruction, where execution should continue if the instruction returns normally. It defaults to +1 (i.e. the next instruction).

The abnormal return point specifies where execution is to continue if the instruction terminates abnormally. It specifies the offset of the exception handler and defaults to the normal return point (i.e. ignore the error) in which case the arguments normally supplied to an exception handler are not supplied.

The argument fields, of which there may be zero or more, refer to the objects to be passed, in the given order, as arguments. The form for arguments allows them to either be fixed (the "name" and "literal" options) or dynamically bound (the indirect through name space options). The number of arguments must be acceptable to the invoked operation or an exception is raised.

The result fields specify where the results of the call are to be placed (i.e. where in the caller's LNS or GNS). The i'th result of the invoked procedure is placed in the location as specified the i'th result specification. If there are M results specified in the instruction and R results actually returned, only min(M, R) of the result locations will be assigned to.

The object rep passed to code_seg$create is nothing more than the specification of a sequence of instructions of this form that are sequentially numbered starting from one. The actual form of rep is unspecified by the thesis as the possibilities are numerous and the details irrelevant for the purposes of this thesis. The only constraint is that all of the possibilities inherent in Figure 2.5 be allowed.

One final note on the execution environment is needed to complete the discussion and this concerns assignment. The form of AESOP instructions only allows accessing and setting LNS (GNS) entries as part of invocation. Copying a reference in one LNS (GNS) slot to another must be accomplished by using this basic mechanism; there is no copy_reference instruction. This is accomplished by acquiring a reference to the LNS (GNS) and then invoking vector$ref with that reference to retrieve the object reference to be copied and placing the result returned by that operation in the appropriate place.

2.4 Object Viewers

This section presents the AESOP mechanism used to implement extended type objects and provide the capability for access restriction and access revocation. The surprising fact is that these three activities, seemingly different at first, can be regarded as being special cases of the same general mechanism. The common property of these activities is that they all permit different users of an object to have different views of that object; extended type objects hide the representation of an object from its users and allow only its interface specifications to show through; access restriction presents an object that does not support all of the operations normally associated with objects of that type; and access revocation causes previously possible operations to no longer be available. Since the common property of these activities is that they present users with differing views of an object, the mechanism used to implement these activities is called the object viewer mechanism.

The basic object viewer mechanism, shown in Figure 2.6, is inspired by the access revocation mechanism proposed by Redell[40]. An object viewer is a triple whose first part is the name of a type manager. This name specifies the type (i.e. the name of a type manager) of the object that is seen when the object viewer is referred through.

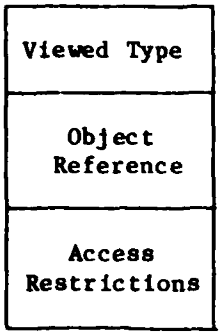


Figure 2.6. An object viewer.

The second field refers to an object. The object viewer is providing a new view of this object; it is the object "seen" in the object viewer. The third field is a bit vector that specifies, by having <u>true</u> in its entries, which operations can not be performed upon the viewed object. The association of an operation with a bit of the access restriction field is unspecified by AESOP; instead it is interpreted by the type manager specified in the type field. A likely association is that the i'th bit controls access to the i'th operation. To illustrate the interactions of the three fields in the object viewer, the next few paragraphs will show how this basic mechanism can be used to achieve the three effects previously mentioned.

First consider access restriction. Suppose a program wishes to pass an object Y of type FOO to another program but does not wish certain operations (e.g. modifications to Y) to be performed upon Y. In this case the first program creates an object viewer with the structure of objects illustrated in Figure 2.7 and passes the name X to the called

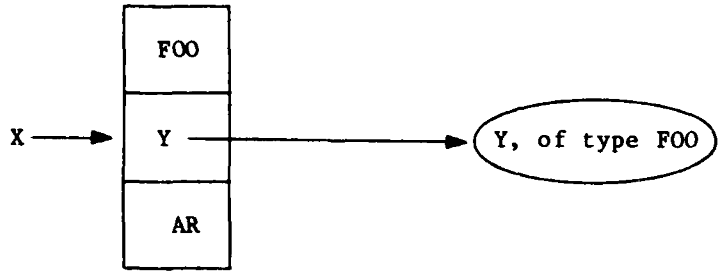


Figure 2.7. Using an object viewer for access restriction.

program. The object viewer in this case specifies that the name X refers to an object of type FOO (the same as that of Y) and the object viewed when X is referred to is Y. The bit vector AR specifies the restrictions upon the way that the called program may use the object it has been passed.

Suppose the second program wishes to pass X to a third program and restrict that program's access to X. It does so by creating a new object viewer as shown in Figure 2.8 and passes the name W along to the third program. When the third program uses W it "sees" the object Y (W "sees" X which transparently "sees" Y) of type FOO so that any

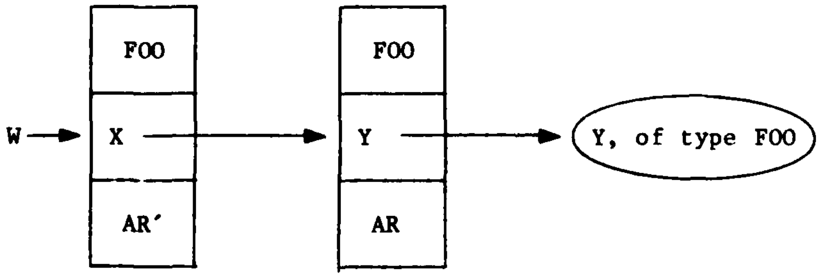


Figure 2.8. A chain of access restriction object viewers.

operations performed using W result in operations on the object Y. The access allowed to the object Y by users of the name W are specified by the minimal access rights specified by AR and AR′, AR <u>or</u> AR′. The effect of object viewers is thus cumulative.

Another use of object viewers is for type extension. Suppose that it is desired to create an extended type object, to be named X, of type FOO from an object named Y of type BAR. Figure 2.9 illustrates how an object viewer would be used to accomplish this. The reference X is a reference to an object of type FOO with the access restrictions specified by AR restricting which operations may be performed on the viewed object (i.e. the object of type FOO). The presence of AR allows the creation of objects of type FOO with varying restrictions on access to that object. This permits, for instance, the file type manager to also provide the "types" read_only_file and stream_file. To have operations performed on the extended type object X, the name X must be passed to the type manager FOO. Thus the object X has been "sealed" against unauthorized manipulation. Only the type manager FOO can "unseal" the object viewer and get at the representation object Y and

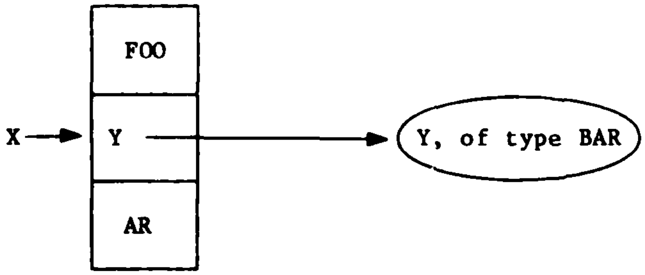


Figure 2.9. Type extension using object viewers.

the access restrictions AR. This allows the type manager to:

1. Determine which operations are permitted on the object X by examining AR.[1]

2. Perform operations on the representation object Y.

No other architectural facilities are necessary to allow the type manager to perform its job.

If the object viewer named by X is to truly "seal" Y, permission to perform the unseal operation must be restricted to the type manager FOO. AESOP enforces this by ensuring that the procedure attempting to perform the unseal operation has been invoked as part of a type manager specified as the viewed type in the object viewer that is being unsealed. Thus the name of the type manager is the "key" that allows the object viewer to be unsealed. If this condition is satisfied, unsealing is allowed; otherwise an exception is raised.

If the ability to perform the "seal" operation were not restricted, any program could seal an object making an object of arbitrary type. When this object is passed to the type manager for that type, the representation of the extended type object might be incorrect and could result in incorrect operation of the type manager. The type manager could protect itself from this kind of misbehavior on the part

[1] This interacts with using object viewers for access restriction. The operations described below handle this case correctly, i.e. access restrictions will be cumulative even in this case.

of other programs;[1] however, this seems to be an added burden on the programmer that is best avoided (especially since the programmer is unlikely to remember to perform the necessary checking at all of the appropriate times). For this reason the seal operation, in which an extended type object is being created, is restricted in the following manner:

1. It may only be performed by a type manager.
2. The field "type" in the created object viewer may only be filled in with the name of the type manager performing the seal operation.

With this restriction, only a type manager may create an object viewer that makes an object look like an object of that type.

Dynamically changing access is achieved by modifying object viewers. Figure 2.10 shows the basic mechanism. X is an object viewer

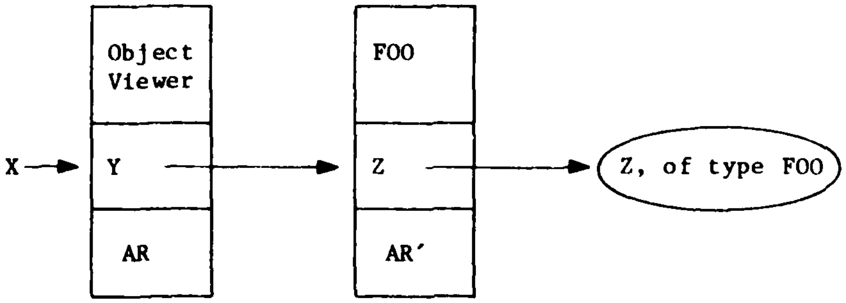


Figure 2.10. Modifying one object viewer with another.

[1] This might be done by including an unforgeable key in the representation for each object (this is similar to the mechanism proposed by Henderson[15]). Such a key might be the name of an object that the type manager never allows to be passed beyond its control.

that has as its type field object viewer and its object reference field names another object viewer, Y. Assume that the access restrictions mentioned in X, AR, impose no restrictions on manipulating object viewer Y. The name X then allows modification of the access restriction and object reference fields in Y. Note, however, that the type field may not be modified since such modification leads to the same problems, in regards to type managers, as mentioned in the case of restricting the ability to perform the "seal" operation. Also, this would change the type of a user's reference dynamically and, although potentially useful, seems far too dangerous to allow. Even so, a program possessing the name X has great power over the contents of the object viewer Y so possession of X must be carefully controlled.

The question arises as to how the object X came into being. It can not be created whenever desired, rather its creation must obey restrictions so that the security provided by object viewers is not compromised. To achieve this, objects such as X are only created as part of the creation operation of other object viewers or as part of an unseal operation performed by a type manager. Thus only the type manager for a given type may manipulate the object viewers that seal objects of that type. Similarly, a program restricting access to an object by creating an object viewer is the only program that can modify that object viewer. In this way the security provided by object viewers depends upon the correctness of the procedures requesting the creation of those object viewers (i.e. if the program gives away a reference to

X, then AESOP can not make any guarantees about the ability to modify Y).

The way of performing access revocation should now be clear. To give revocable access to the object Z in Figure 2.10, the object viewer Y should be created and, at the same time, X should be created. The name Y should be passed to the program that is to be given revocable access and the name X should be remembered. Later when it is desired to revoke some, or all, access to Z, X provides the means to do so. If complete and permanent revocation is desired, X can be used to destroy object Y, making any outstanding references to it invalid. If partial (or non-permanent) revocation or if access enhancement is desired, X may be used to modify the access restriction field in Y appropriately.

Up to this point this section has presented the basic object viewer mechanism and examples of how to use it in solving various problems. The next few paragraphs will present a few rules for using object viewers.

Given an object reference X, it is necessary to resolve that reference to determine the referenced object, the type of the referenced object ani any restrictions on the use of the referenced object. To find this information, it is necessary to follow a chain of object viewers until one is found that is being used for type extension. Let NAME be the record:

record(type:tm, object:<u>any</u>, ar:vector[boolean])

Resolving a reference X involves filling in NAME for X. If X does not refer to an object viewer then NAME = (type-of-X, X, []) (i.e. no type extension and no access restrictions). Otherwise X names an object viewer, call it ov. In this case set NAME = (ov.type, X, ov.ar). Let ov′ = ov.object. Now as long as NAME.type = ov′.type and ov′ is an object viewer do:

1. NAME.object = ov.object
2. NAME.ar = NAME.ar or ov′.ar
3. ov = ov′
4. ov′ = ov′.object

When done, NAME will contain the desired information (i.e. object referenced, its type and the restrictions on the use of that object).

Now that the concept of object viewers is understood, the operations provided by the object_viewer type manager, abbreviated by ov, are presented in Figure 2.11. The seal operation creates an

```
ov$access(o) returns(ar:vector[boolean])
ov$extract(o) returns(sealed_object, ar:vector[boolean], revoker:ov)
ov$modify(revoker:ov, obj, ar:vector[boolean])
ov$restrict(viewed_object, ar:vector[boolean]) returns(X, Y:ov)
ov$same_end_objects(o(1), o(2)) returns(b:boolean)
ov$same_names(o(1), o(2)) returns(b:boolean)
ov$seal(sealed_object, ar:vector[boolean]) returns(X, Y:ov)
ov$type(o) returns(t:tm)
```

Figure 2.11. The operations on object viewers.

extended type object, X, whose fields are set to the name of the current type manager, sealed_object and the value of ar. The restrict operation creates an object viewer, X, that permits restricted access to viewed_object as specified by ar. In both these operations, Y permits the object viewer X to be manipulated. The access and type operations permit the determination of the type of an object reference o and the access restrictions on using it. The extract operation is used by the type manager that implements o to follow a chain of object viewers being used as access restrictors and return the effective access (ar) allowed to o in ar, the sealed object in sealed_object and a revoker that permits modification of the object viewer sealing the extended type object o in revoker. The type manager may then interpret ar to enforce access restrictions and use sealed_object as the representation object of o. The modify operation allows for the modification of an object viewer, call it O, referenced in the object viewer revoker (i.e. revoker permits modification to O) by setting O's object field to obj and access restrictions field to ar. Obj must be nil or of the same type as O's viewed type field or an exception is returned.

The user of AESOP may need to ask the question: do these two object references refer to the same object? The object viewer type manager answers this question. The same_names operation returns true if and only if o(1) and o(2) refer to the same object; that is, it returns true if they refer to the same non-object viewer object directly or they refer to the same object viewer. If it returns true then the references o(1) and o(2) will always refer to the same object with the same access (i.e. same_names does "pointer" equality). The same_end_objects operation returns true if and only if o(1) and o(2) provide possibly differing views of the same object; that is, it returns true if and only if the chains of object viewers named by o(1) and o(2) eventually converge. If it returns true then o(1) and o(2) currently refer to the same objects, with possibly differing access restrictions, but there is no guarantee that this equality will continue at any time in the future due to the ability to modify the object and access restriction fields in non-common object viewers.

Each of the built-in type managers will enforce access restrictions by associating one bit of the access restriction field in object viewers with each operation they provide. In this way, access to built-in operations can be individually controlled.

2.5 Operating System Related Features

This section describes the features provided by AESOP to meet the goal of elimination of the operating system. In addition this section describes architectural features usually hidden by operating systems that are needed to make the architecture complete (e.g. what it means to initialize the processor).

One operating system feature, protection, is provided by the object viewers described in the preceding section. This section describes how the operating system notions of process, interprocess communication/synchronization and storage management are provided.

These features have been chosen for inclusion in AESOP as they meet the high level criterion, presented in chapter one, of being language and application independent. The motivation for processes and inter-process communication/synchronization being in the architecture have been presented in chapter one. Storage management must be provided at the architectural interface since built-in operations consume storage (e.g. vectors). Given that storage management is necessary (see chapter one), the only alternative to an architecturally supplied facility is a layer of software between users and those architectural features that consume storage. This software is, however, nothing more that a primitive operating system and so is rejected.

Every system needs to communicate with the rest of the world and to start up and shut down. If they are not provided by the architecture, in however a primitive a form, they can never occur. Thus AESOP must provide an I/O facility and define the effects of starting and stopping an AESOP processor.

2.5.1 Processes

AESOP allows the user to have multiple processes. An AESOP process is the execution of a procedure in parallel with all other processes running on the architecture. The architecture makes no guarantees as to how this parallelism is achieved so that any implementation from multiprocessing a single processor to providing a single processor per process is acceptable. Some of the operations on processes are presented in Figure 2.11 (control flow instructions have been described in section 2.3 and more are presented in appendix A).

A new process, pr, is created by the create operation. The process pr begins execution by executing the instruction:

```
proc$call_with_gns(gns, p, arg(1), ..., arg(n))
```

so that the execution of pr is exactly the same as the execution of p with the given arguments. The process pr terminates when p returns (any returned values are ignored). The argument lca specifies the default storage area, the place where newly created objects are placed by default when pr creates them.[1] Note that the creating and created process share access to the objects gns, arg(1), ..., arg(n) and all objects accessible through them so that the problem of shared data must be addressed.

```
process$create(p:proc, gns:vector, lca:storage_area, arg(1), ...,
   arg(n)) returns(pr:process)

process$max_priority() returns(prior:integer)

process$schedule(pr:process, prior:integer, limit:integer, event:ec)

process$start(pr:process)

process$status(pr:process) returns(s:integer, cpu:integer,
   prior:integer, ec:vector[event_count], limit:integer,
   event:event_count, other)

process$stop(pr:process)
```

Figure 2.12. Some of the operations on processes.

[1] Storage areas will be discussed later in section 2.5.3.

Chapter one argued that the programmer requires some limited control over the processor resources consumed by processes. Four operations are provided for this purpose. The stop operation causes a process to stop execution until a start operation is executed on that same process. (By convention every newly created process is created in the "stopped" state.) To decide when to stop a process, it must be possible to determine the status of that process. The status operation returns the status of process pr as follows:

s indicates whether pr is runnable (e.g. is, in principle, consuming processor cycles), stopped, blocked waiting for some event or terminated.

cpu indicates, in implementation defined units, how much CPU time pr has consumed.

prior is pr′s current scheduling priority (see below).

ec is an array of event counts that pr is blocked on.[1]

limit is pr′s execution time limit.

event is an event count that is to be incremented whenever limit is exceeded.

other is an implementation defined object that an implementation may use to return other status about pr (e.g. how many page faults it has taken).

With this information intelligent decisions regarding pr should be possible.

[1] An event count is AESOP′s interprocess synchronization primitive and is discussed later in this section.

The ability to stop and start a process is a coarse means of control and a finer grain of control may be needed. In choosing a finer grain of control for processes, AESOP has to make a guess at what is needed since there are no personal computers available today. A number of systems provide the user with the ability to spawn multiple processes (Tenex and Unix for instance). None of these, however, gives the user much control over the spawned processes beyond the ability to destroy and stop them.

Reed[41] has shown how a simple low level scheduling mechanism can be used to construct complicated high level scheduling mechanisms. A similar approach has been taken in Hydra[27]. AESOP will provide a mechanism for controlling processes that is based upon Reed's work. Every process will have associated with it a priority (with a maximum of process$max_priority()) and a CPU time limit. Within the class of runnable processes the process with highest priority will always be run.[1] A process will run until it is preempted by a higher priority process entering the runnable queue or until it exceeds its CPU time limit. When a process exceeds its CPU time limit, it enters the "stopped" state and an event count is incremented so as to notify some other process(es) that this event has occurred.

The schedule operation sets the priority and CPU time limit of a process. It is the fourth, and last, operation provided by AESOP to control processes. After any of these four operations is performed on a

[1] In the case of ties, a round robin policy will be used.

process, the set of all runnable processes is examined and the highest priority process(es) are allowed to execute. Note that if the priority of a process is set to max_prior(), that process is guaranteed to get some processor resources due to the round robin tie breaking policy. This permits the construction of "watch-dog" processes that, for instance, watch for other processes that seem to be in an infinite loop.

2.5.2 Interprocess Communication and Synchronization

AESOP′s processes can communicate with each other through the objects that they share access to. Those processes not only need to communicate but also to synchronize their activities including synchronizing access to those shared objects. For these reasons, AESOP must provide an interprocess synchronization mechanism.

Many mechanisms have been proposed for achieving interprocess synchronization including block/wakeup[44], semaphores[10], messages[14,20], monitors[19], serializers[18] and event counts/sequencers[42]. All of these are roughly equivalent in terms of expressive power; they all permit processes to synchronize their activities. Choosing amongst these is, to a first approximation, a matter of personal taste.

AESOP makes the choice of event counts/sequencers as the basic means for interprocess synchronization.[1] Event counts have been chosen not for their expressive power but, rather, because they capture the

[1] See Reed[42] for detailed semantics of what follows.

essence of the interprocess synchronization problem while minimizing the communication aspects. This is desirable since AESOP provides communication through shared memory, only a synchronization mechanism is needed. The operations on event counts and sequencers are presented in Figure 2.13.

An event count is a counter that may only be incremented. A new event count, initialized to have value zero, is returned by the create operation. The current value of an event count is returned by read. The increment operation increments an event count. The await operation causes the executing process to cease execution until ec\$read(e(i))≥c(i) for some i and returns that i. When a process executes an await operation it becomes unrunnable so that other, lower priority processes may run. When ec\$read(e(i))≥c(i) for some i, this process becomes runnable again so that it will preempt other, lower priority, processes. This is the interaction between the process scheduler and the

```
event_count$await(e(1):event_count, c(1):integer, ..., e(N):event_count,
   c(N):integer) returns(i:integer)

event_count$create() returns(e:event_count)

event_count$increment(e:event_count, n:integer)

event_count$read(e:event_count) returns(count:integer)


sequencer$create() returns(s:sequencer)

sequencer$take(s:sequencer) returns(ticket:integer)
```

Figure 2.13. The operations on event counts and sequencers.

synchronization mechanism. One of the reasons that event counts have been chosen for AESOP is that this interaction can be efficiently implemented (see chapter four).

AESOP does not predefine any event counts. However, a particular implementation may choose to predefine some event counts such as an event count that serves as a calendar clock.

Event counts allow two processes to synchronize their parallel activities through event counts that they both can access (e.g. they have a common data base or were passed the event counts when they were created). However, an additional mechanism is needed to permit general access to shared data. The sequencer mechanism allows this. A sequencer is an object that returns, via the take operation, a monotonically increasing sequence of integers. The first take operation on a sequencer returns, by convention, one. Controlled access to shared information can be achieved by the appropriate use of sequencer$take and ec$await as explained by Reed.

2.5.3 Storage Management

If there were infinite amounts of equally accessible storage on AESOP, there would be no need for user visible storage management. Implementation realities force the consideration of a finite, multi-level memory.

Two problems are of concern: storage quotas and efficient inter-object references. Storage quotas are used to prevent a buggy or untrusted program from consuming all available storage. If it were allowed to do so, the system would stop as no program could run due to lack of storage. More importantly, it is difficult to create a daemon program that notices the situation and corrects it since any such program would need storage to run and a great deal of privilege to discover which programs were causing the problem and to correct the problem. Outside intervention is traditionally used in language systems to fix the situation (i.e. by returning to the operating system interface to correct the problem). This solution is unacceptable in AESOP as there is no operating system. Leaving the AESOP system to solve the problem is unacceptable as it would violate the constraint imposed on AESOP, in chapter one, that AESOP must be complete in and of itself; there must be no need to leave AESOP to solve a problem, including this one. As a result storage management must be present in the AESOP interface.

An AESOP machine will likely consist of some amount of primary memory and a larger amount of slower, secondary memory. References to objects in primary memory are likely to be more efficient than references to objects in secondary memory. Moreover, there is likely to be a high cost associated with moving objects between primary and secondary memory. As a result, an implementation must attempt to maximize the chance that an object reference will refer to an object that is in primary memory. With no other information this is likely to be difficult, so some information supplied by the user through the architectural interface to the implementation is appropriate.

Both of these goals will be met by the introduction of a new type of object, storage areas. A storage area is a quota pool, a pair of positive integers reflecting the amount of used and free storage in the area. Whenever an object is created, it draws storage from some storage pool causing the amount of free storage in that pool to change. To account for space hungry programs, it is only necessary to arrange that it draw storage from a storage pool of suitable size.

An object created in storage area s is said to be contained in s. For the purposes of efficient multi-level memory management, all objects in a storage area s are assumed to frequently refer to each other and only infrequently refer to objects in other storage areas. That is, the objects in a storage area should exhibit locality of reference.

Objects in some storage areas may tend to refer frequently to objects in a second area. If this relationship is made explicit, an implementation of AESOP may be better able to handle references from area s to area s′ in an efficient manner and use this information to improve the performance of the multi-level memory. For the reason, the <u>close</u> relation on storage areas is defined. If area s is <u>close</u> to area s′, objects in s frequently refer to objects in s′. The close relation is neither reflexive, symmetric nor transitive.

The operations on storage areas are presented in Figure 2.14. The create operation creates a new storage area, new_area, by withdrawing size units of quota from parent_area. The close operation indicates that area s is close to area s′. If, at a later time, the user should decide that the close relation is no longer appropriate, the not_close operation should be performed. The size operation returns the size and current amount of free storage in a storage area.

Since every object must be created by drawing storage from some area, the various operations of the built-in type managers that create objects (e.g. vector$create) must, in principle, take an extra argument that is the area the newly created object draws quota from. The constant supplying of an area argument can get quite tiring. For this reason, every process has a default storage area associated with it that is used whenever a storage area argument is not supplied to a built-in create operation. The default storage area is initially set for a process by the lca argument supplied to the process$create operation, may be reset by the

```
process$set_default_area(s:storage_area, pr:process)
```

```
storage_area$close(s:storage_area, s′:storage_area)

storage_area$create(size:integer, parent:storage_area)
   returns(new_area:storage_area)

storage_area$not_close(s:storage_area, s′:storage_area)

storage_area$size(s:storage_area) returns(size:integer, free:integer)
```

Figure 2.14. The operations on storage areas.

operation and may be discovered by the

```
process$get_default_area(pr:process) returns(s:storage_area)
```

operation.

Not all of the storage in AESOP will be consumed by AESOP objects, some will be consumed by the implementation of AESOP. Because the needs of an implementation may be highly variable, it is advantageous to shield user programs from this variability since it may lead to programs whose correctness is implementation dependent. Thus the storage the implementation uses must not appear to the users of AESOP as consuming part of a storage area. Instead, an implementation will draw storage from an architecturally hidden pool of storage. The user shares use of this pool with the implementation by creating storage areas out of it by using the built-in procedure:

```
master_alloc(size:integer) returns(s:storage_area) signals(cant)
```

The exception cant is raised if the implementation should decide, for whatever reason, that a new storage area of the specified size can not be created now. This procedure is passed to the user at system initialization time. Since this procedure essentially permits the creation of storage areas out of nothing, the user must carefully control its use so that the problem of storage hungry programs does not arise.

2.5.3.1 Storage Reclamation and Object Deletion

AESOP provides a heap oriented storage semantics - objects are created in a storage area and remain there, drawing storage from that storage area, until the object is no longer needed. This section discusses the determination that an object is no longer needed.

An object is no longer needed when it is no longer accessible (i.e. no longer referenced by a needed object). This is the case since it is impossible for any references to that object to ever exist in the future due to the capability-like nature of AESOP's naming mechanism. Once an object has been determined to be no longer needed, the storage consumed by that object is returned to the free pool of the storage area that contained that object. Thus AESOP constrains an implementation to reclaim the storage used by unneeded objects. This allows for the carefree reclamation of storage.

The automatic reclamation of storage may not always be sufficient to meet the user's needs when the problem of buggy or untrusted programs is considered. Although the basic storage area mechanism allows the amount of storage usable by such programs to be controlled, the user may wish all of the objects created by that program to be immediately discarded. The user may be experiencing a space shortage and simply need that storage or the user may not believe that whatever objects that program created are correct and does not ever want to reference them. As a result, the user has a need to explicitly delete the objects that a program has created.

Explicit object deletion creates the problem of dangling references to deleted objects. The system is responsible for ensuring that all outstanding references to a deleted object are invalidated. The invalidation consists of marking those references as "reference-to-deleted-object".[1] Any future attempt to use one of those references will result in an unexpected_deleted_operand exception so that the user must be prepared for his programs to fail in this way if they happen to reference a deleted object. To allow for explicit object deletion, the following operations are provided. The operation

```
storage_area$delete_all(s:storage_area)
```

deletes all of the objects that are in area s. In addition, type managers may provide operations to delete their objects. Most of the built-in type mangers do just this (see Appendix A).

2.5.4 I/O

To be useful, AESOP must provide the means for the user to write programs that communicate with I/O devices (e.g. a network, a terminal or a disk drive). Due to the diversity of I/O devices, it is undesirable to specify device specific I/O. Rather, this section will present a general framework in which I/O in AESOP will occur.

[1] An implementation problem is to find all of these references. A means for doing this is discussed in chapter three.

2.5.4.1 Object Oriented I/O

Since AESOP is an object oriented architecture, it is reasonable to consider having I/O closely follow that model. Object oriented I/O would allow I/O devices to access all of the objects in the object memory provided by AESOP as objects (i.e. through the type manager corresponding to the object). Although appealing, this approach must be rejected.

This approach implies that the I/O devices must not only be prepared to deal with the built-in objects of AESOP but also with extended type objects. This can be done in two ways. First, the AESOP implementation could define an interface that allows a device to access all objects. This, however, is just pushing the problem of I/O into the implementation since I/O devices really only deal with bits and so is no solution.

The second solution is to allow I/O devices to be processors capable of directly accessing and using all AESOP objects, including type managers. This fails for three reasons. First an AESOP implementation has been transformed from what can be a uni-processor implementation to an implementation that inherently contains multiple processors, the main AESOP processor plus I/O devices. This means that the algorithms used to implement AESOP become more complicated to cope with this parallelism. This extra complexity may very well mean that an AESOP processor becomes uneconomical. Second, I/O devices may become prohibitively expensive to attach to an AESOP system due to their complexity. Third, this approach violates one of the prime goals of AESOP, the hiding of implementation issues by the architecture, since I/O devices must be given the specification of the implementation of AESOP so as to be constructed in a manner that is compatible with the rest of the AESOP system. For all three reasons this approach is also rejected. Since both solutions are unsatisfactory, object oriented I/O is rejected for AESOP.

2.5.4.2 Bit Oriented I/O

The alternative to object oriented I/O is a traditional approach in which I/O devices are allowed to access memory as a collection of bits and do not see any of the higher level features of AESOP. This sort of interface can be provided in (at least) two ways: a channel program interface as is done for IBM 360/370 systems or a device register approach that models devices as a collection of registers that are directly addressed and manipulated by user programs as in DEC's PDP-11 family of computers.

The channel program approach consists of supplying a program to an auxiliary processor, the channel, that actually moves data between the I/O device and primary memory. This approach has the advantage that multiple commands may be given to a device within a single channel program, thus limiting the need for the CPU to intervene in I/O. The device register approach, on the other hand, requires that each command for a device be initiated individually by the processor. The channel program approach has the disadvantage of requiring the creation of a new object, the channel program, for I/O initiation.

On balance, the channel approach is rejected since the economics that created it, expensive processor cycles, no longer exist - LSI and VLSI make processor cycles cheap. Also, the complexity of the channel program objects themselves is unattractive.

AESOP will present I/O devices at its interface as an array of device registers that contain objects of restricted type. The operations on I/O devices are given in Figure 2.15. A newly attached I/O device is made known to AESOP by executing a new_device operation that takes the location, addr, of the device in an implementation defined address space and a specification, spec, of the interface to that device as arguments.

A device interface consists of a sequence of registers each of which is either a status, event count or buffer register. A status register consists of a sequence of bits through which the user and the

```
io$new_device(addr:integer, spec:vector[integer]) returns(dev:io)

io$read_register(dev:io, i:integer) returns(register)

io$set_status_abcd(dev:io, i:integer, bv:vector[boolean],
   offset:integer, length:integer) returns(status:vector[boolean])
   % This is a set of operations derived by setting
   %  a, b, c and d to 0 or 1

io$set_register(dev:io, i:integer, new)
```

Figure 2.15. The operations on I/O devices

I/O device communicate by setting and reading status registers. An event count register contains a reference to an event count that the I/O device may cause to be incremented so as to communicate the occurrence of some event (e.g transfer complete) to a AESOP waiting process.[1] Buffer registers contain the names of vector[boolean] objects that the I/O device reads and writes as buffers. Once the user assigns a vector to an I/O device's buffer register, the contents of the vector are undefined until the device indicates to the user that the I/O is complete. During this time, the user should not manipulate the vector nor assign it to another buffer register. The effect of such actions is not specified. However, if the device will only read the contents of the bit vector, the user may also read the bit vector and get the correct contents of the vector.

The device registers may be manipulated by a collection of operations provided by the io type manager. The contents of the i'th register are returned by the read_register operation which returns either an event count, a vector[boolean] used as a buffer or a newly created vector[boolean] that contains the value of a status register. The contents of the i'th register is set to new by the set_register operation which raises an exception if the type of new is not acceptable as the contents of the i'th register of dev.

[1] This is similar to the technique used in the Venus operating system[29] where semaphores were used as the synchronization primitive.

The set of operations set_status_abcd permit the manipulation of individual bits of a status register. The new value of the status register is calculated by performing the boolean operation indicated by abcd on a subvector of the status register (as indicated by offset and length) and the vector bv. In particular, bit j+offset-1 of the status register, for j from 1 to length, is modified to have the value given by the following table:

x\y	0	1
0	a	b
1	c	d

where x is the value of the j'th bit of bv and y is the value of the j-offset+1'th entry of the status register. This operation is defined to be atomic with respect to all other io$set_status_abcd operations on the given status register.[1]

It should be clear that this is only a framework in which I/O in AESOP will occur; it is not a detailed specification of the inner workings of I/O as such details are very device specific. Rather, this definition of the io type manager should give the programmer the ability to control I/O devices given the particular characteristics of that device. The actual bits that a device reads/writes will be converted

[1] It is not clear that this atomicity is absolutely necessary, but defining things in this manner may permit some devices to be handled in a cleaner manner. Also, since this definition does not specify the effects of status register manipulation by the device, it is easy to provide as will be seen later in the thesis.

from/to AESOP objects by operations supplied either by the user or by the architecture in a manner described by Herlihy[16] and Sollins[48].

2.5.5 System Initialization and System Shutdown

No system runs forever. Instead they are occasionally shutdown and then restarted later. This section briefly discusses the issues of starting up and shutting down an AESOP processor.

2.5.5.1 System Initialization

System initialization is the act of bringing a system from a state of no activity, with no processes running or runnable, to the state of having the system in normal operation. In the case of AESOP, system initialization also brings out the problem of which objects in the system are accessible when the system is restarted.

AESOP has a distinguished object, a vector named ROOT, to meet these needs. It is distinguished in that it always considered a needed object and is never garbage collected. Moreover it is an object that can never be deleted. The set of needed objects in a newly started AESOP is precisely the set of objects that are accessible from ROOT. ROOT, by convention, will have the following form:

```
ROOT[1] = p:proc

ROOT[2] = lca:storage_area

ROOT[3] = gns:vector
```

ROOT[4] is reserved for AESOP’s use

ROOT[i] for i>4 can be anything.

System initialization consists of the processor executing the following built-in program:

```
pr:process := process$create(p, gns, lca, ROOT,
   master_alloc)

ROOT[4] := pr

process$schedule(pr, process$max_prior(), nil, nil)

process$start(pr)
```

The effect of this program is to create a new process, pr, that executes procedure p with gns as its GNS and lca as its default storage area. The procedure p receives two parameters: ROOT and the built-in procedure master_alloc. ROOT[4] is then set to pr so that pr may manipulate itself if needed. Finally pr begins execution with the maximum possible priority. The process pr is responsible for performing any other initialization activities that the user of AESOP feels are needed.

2.5.5.2 System Shutdown and Crashes

The companion activity to system initialization is system shutdown. An unfortunate but probably inevitable event is a crash of the system.

Shutdown in AESOP is accomplished by placing all processes in the stopped state, thus stopping all activity in AESOP. When AESOP shuts down, the implementation of AESOP is responsible for ensuring that all objects in the system will still be around if the processor is now powered down. Thus the act of stopping the last runnable process on AESOP, and thus shutting down AESOP, must be accomplished, by the implementation of AESOP, by placing all objects in AESOP on non-volatile storage.

A system crash is an unscheduled stopping of the system. When the system crashes there will, in general, still be runnable processes and objects that reside in volatile storage. Ideally the effect of such an event would be specified by the architecture. Such a specification might be of the form: all architecturally defined operations are atomic; that is, every built-in operation either occurs or does not occur and if it does occur, no crash can cause the action to be (partially) undone. Unfortunately, such a specification is very difficult to meet in an economical manner. The whole area of robustness to system failures is an important one but its solution in the economically constrained environment of a personal computer has not been achieved and it is beyond the scope of this thesis to find a solution. Thus the effect of system crashes is left unspecified in this thesis. Instead, each implementation of AESOP will specify the effect of system crashes in an ad hoc manner until an acceptable solution to the problem is found.

2.6 Conclusions

This chapter has presented AESOP - its built-in types and the operations on them. There is nothing sacred about most of the operations provided so that the reader should feel free to modify the set of operations to provide any desired operations. What is important are the variety and capabilities of the built-in type managers. The various features that were argued, in chapter one, as being necessary have been presented. This is the first part of the demonstration that AESOP is a high level architecture. Chapter five will complete that proof by showing how to use AESOP. First, however, the next two chapters will show one possible implementation of AESOP. This will give the reader reason to believe that AESOP can be efficiently implemented. This belief will give the reader confidence that the uses of AESOP presented later in the thesis are, in fact, reasonable to imagine.

AD-A083 433   MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTE--ETC F/G 9/2
THE ARCHITECTURE OF AN OBJECT BASED PERSONAL COMPUTER.(U)
MAR 80 A W LUNIEWSKI N00014-75-C-0661
UNCLASSIFIED MIT/LCS/TR-232 NL

2 OF 3

AD
A083433

Chapter Three

The Implementation of Logical Storage Management

Chapter two described AESOP, an architecture that provides an object based environment for program execution. This chapter, and the next, present an implementation of AESOP to demonstrate that simple and efficient implementations of AESOP exist. Since there are many possible implementations of AESOP, this particular implementation should be regarded only as an existence proof.

3.1 An Introduction to the Implementation

The fundamental goal of any implementation of AESOP is to implement the particular semantics presented in chapter two. In a high level sense nothing else matters, but from a practical point of view efficiency is a major concern.

An implementation should be both space and time efficient. An ideal implementation would use no storage beyond that needed to hold the user's data and would allow the user's programs to execute at a speed comparable to that of the underlying hardware. In practice these goals are unrealizable; space and time overhead is inevitable in any implementation. This implementation of AESOP is no different.

This implementation of AESOP has, in general, been designed to save storage at the possible expense of execution efficiency. This choice has been made since personal computers tend to have a fairly small amounts of storage while the user generally does not have enough work to keep the machine busy (i.e. there are instructions to waste but not memory). The impact of this assumption will primarily be seen in the next chapter when physical memory management is discussed.

An AESOP implementation must manage the logical memory defined by AESOP, map AESOP's logical memory onto a physical memory and implement AESOP's basic types. This chapter discusses logical memory management. This is primarily a problem in performing storage allocation and reclamation. It is difficult due to the need to deal with a large number of objects. Chapter four discusses how to map AESOP's logical memory onto a physical memory. This involves allocating/reclaiming the storage needed by AESOP's logical memory and mapping logical memory addresses into physical memory addresses. This is a hard problem due to the need to achieve reasonable space and time efficiency. The next chapter also shows how to implement AESOP's basic types. For the most part these are trivial. However, four types are of concern: processes/event counts, code segments, I/O and storage areas. Processes must be cheap so that they be used freely. Code segments must encode AESOP instructions to permit the efficient execution of AESOP code. I/O is important as it specifies the interface between an AESOP system and the outside world. Storage areas embody the logical memory management algorithms presented in this chapter.

This chapter discusses the management of AESOP's logical memory. AESOP defines a logical memory in which a large number of objects are constantly being created, used and then discarded. Logical memory management is keeping track of these objects and reclaiming their storage when they are no longer accessible.

One assumption about the underlying hardware is crucial. AESOP is assumed to be implemented as a uni-processor so that the algorithms presented here need not be concerned with parallelism at the implementation level, thus simplifying them considerably. It is left as an exercise to modify these algorithms, or develop new ones, that are appropriate for a multi-processor implementation.

The logical memory management algorithms presented in this chapter are based upon the work of Bishop[5] on garbage collecting large address spaces. Section two briefly reviews his storage management ideas. If an architecture implemented using Bishop's techniques is used incorrectly, poor performance may result. Section three presents a particular pattern of use of AESOP, the subsystem model, that, when followed, avoids these problems. It is the basis, in this thesis, for believing that Bishop's techniques are practical. Section four presents the garbage collection algorithm used for AESOP. It is unique in that it combines Bishop's basic techniques with Baker's real-time garbage collector[3]. Section five discusses object creation and deletion. As it turns out, the basic garbage collection algorithm of section four does not collect cycles of unneeded objects that span multiple storage areas. Section six presents an algorithm to remedy this. This chapter concludes with a discussion of an efficient implementation of the stack of LNS's and control information associated with invocation in AESOP.

3.2 Bishop's Mechanism for Logical Memory Management

The mechanism proposed in this thesis to reclaim storage occupied by inaccessible objects, garbage collection, is based upon the ideas of Bishop[5]. This section reviews the relevant aspects of his work and points out some potential problems with his scheme.

An object reference in Bishop's scheme denotes an object by giving its location in a large, linear address space and its type. The address space is divided into a number of storage areas (a linear, connected subset of the address space). Storage areas serve two purposes. First, they serve as quota pools. Every object is created in a storage area, is located entirely within that storage area and consumes the storage of that area. Second, storage areas serve to limit the bounds of garbage collection. The goal is to garbage collect a single storage area without worrying about the contents of other storage areas. If an area is small enough, a little smaller than the size of primary memory according to Bishop, then garbage collecting that area is efficient since secondary memory need not be accessed randomly or frequently to perform the garbage collection.

There is a conflict between the ability of objects to refer to any other object in the logical memory and the desire to garbage collect a storage area without examining other storage areas. This conflict is the dangling reference problem. Consider Figure 3.1. Assume that there

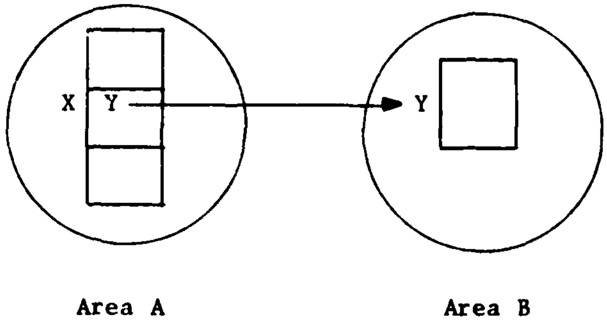


Figure 3.1. An example of inter-area references.

are no references to Y except for the single reference from X. If an attempt is made to garbage collect area B without examining other areas, the object Y will seem to be garbage and its storage reclaimed. At this point X will contain a dangling reference to Y. Any attempt to use it is in error and the implementation must prevent such use. Bishop prevents such dangling references via two mechanisms: inter-area links and inter-area cables.

3.2.1 Inter-area Links

An inter-area link from area A to area B, referring to object X, asserts that there exists at least one reference in A to X. Furthermore, that reference is an indirect one through the inter-area link itself, which resides in A. Each area, A, keeps all of the inter-area links to A on a list. The garbage collector, when garbage collecting A, uses this list to find all of the objects in A that are referenced from other areas via inter-area links. In this way no dangling pointers result since all accessible objects are marked as used.

The maintenance of the list of inter-area links is very simple. Whenever an executing program attempts to store a pointer into some area, the implementation checks to see if that pointer refers to an object in another area. If it does, an inter-area link is created and added to the list of incoming links for the referenced area.[1] The reference itself is modified to refer indirectly through the newly created inter-area link.

The use of inter-area links has three drawbacks. First, it consumes storage at a large pace since every inter-area reference requires the creation of an inter-area link. Second, the scheme slows execution when an inter-area reference is used since two memory references are required (one to the inter-area link and one to the target object). Third, copying pointers may be expensive since an inter-area link may need to be created and added to the list of incoming references. For all three reasons, the use of inter-area links is discouraged by Bishop.

3.2.2 Inter-area Cables

As an alternative to inter-area links, Bishop proposes inter-area cables. An inter-area cable from area A to area B allows pointers in objects in area A to refer directly to objects in area B without the need for using inter-area links. In this case, area A is said to be

[1] It is possible for multiple references from area A to area B to share links at the expense of searching the set of all outgoing inks from A to B when a new inter-area reference is created.

cabled to area B. Cables make references from area A to area B cheap both in terms of storage (no inter-area link is created) and in terms of time (there is no overhead for copying such pointers or for using them).

Although cheaper than inter-area links, inter-area cables are not free. Whenever an object reference to an object in storage area B is copied into a storage area A it is still necessary to make sure that an inter-area cable exists from area A to area B[1] by searching a table of outgoing cables from area A. I˙ a cable does not exist, a cable must be added to that table to handle thꞷ reference to area B. Another cost associated with cables concerns the scope of garbage collection. Whenever area B is garbage collected, all of the areas cabled to area B must be garbage collected at the same time. This is necessary in order to avoid the dangling reference problem mentioned previously since those areas now have direct references into area B. This increases the amount of storage that must be garbage collected at one time. In addition, area B must maintain a list of all areas cabled to it so that it can perform this garbage collection.

Two conclusions, one good and one bad, may be drawn from this discussion. The good conclusion is that the basic mechanism does work. It allows for arbitrary references from one object to another; it prevents dangling references; and it permits efficient garbage collection. Unfortunately, the use of inter-area links and cables must

[1] Assuming that a decision has been made to handle references from area A to area B by cables and not by links.

be minimized for efficiency reasons. As the rate of creation and use of links and cables increases, the performance of the system will slowly (or perhaps not so slowly!) degrade.

3.3 The Subsystem Model of Storage Use

This section proposes a pattern of use of AESOP that minimizes the use of the expensive mechanisms of Bishop's logical memory management scheme. The subsystem model achieves this. It is hypothesized that much of the use of AESOP will follow this pattern.

Under the subsystem model, the computations being run on AESOP tend to be organized into one or more collections of one or more processes. Each such collection of processes, called a subsystem, has a set of data that all of the processes in that subsystem tend to use. Each process in a subsystem has some private data. The processes in a subsystem communicate with other subsystems via message passing.[1]

A subsystem provides a service such as a file system. A subsystem is a more useful model for services than, say, procedures since it makes explicit the fact that there are many parallel users of the service. A similar model, the guardian model[49], has been proposed for distributed systems to, in part, also make this parallelism evident. Subsystems differ from guardians in that subsystems may share memory but guardians may not.

[1] This is true at a high level, although the implementation of the message passing will involve shared memory.

The patterns of memory sharing under this model are fairly simple. A process in a subsystem frequently uses the data local to that process. The data common to the subsystem is accesses less frequently, although its use is not insignificant. Occasionally a process in a subsystem will need to communicate with other subsystems and, at such times, it will access some memory that is common to those two subsystems. Access to libraries, which contain objects such as language support routines that are used by many subsystems, will also be fairly frequent.

To put this model into the terms of storage areas, see Figure 3.2 which depicts four types of storage areas. An LCA (Local Computation Area) storage area contains the private storage for a process. The subsystem area contains the objects that tend to be shared by the processes of the subsystem possibly including the data that the subsystem manages in providing its service. The library areas contain those objects that are shared by many subsystems such as procedures and type managers. The communication areas allow processes in different subsystems to communicate with each other.

A process can refer to the objects in its LCA without the need for links or cables. The cable from each LCA to the corresponding subsystem storage area allows a process to efficiently access objects in its subsystem area. Each LCA is also cabled to library areas to allow processes to use the objects in those library areas with little penalty. Cables are important in these two cases since a large fraction of the references made by a process outside of its LCA will fall into one of

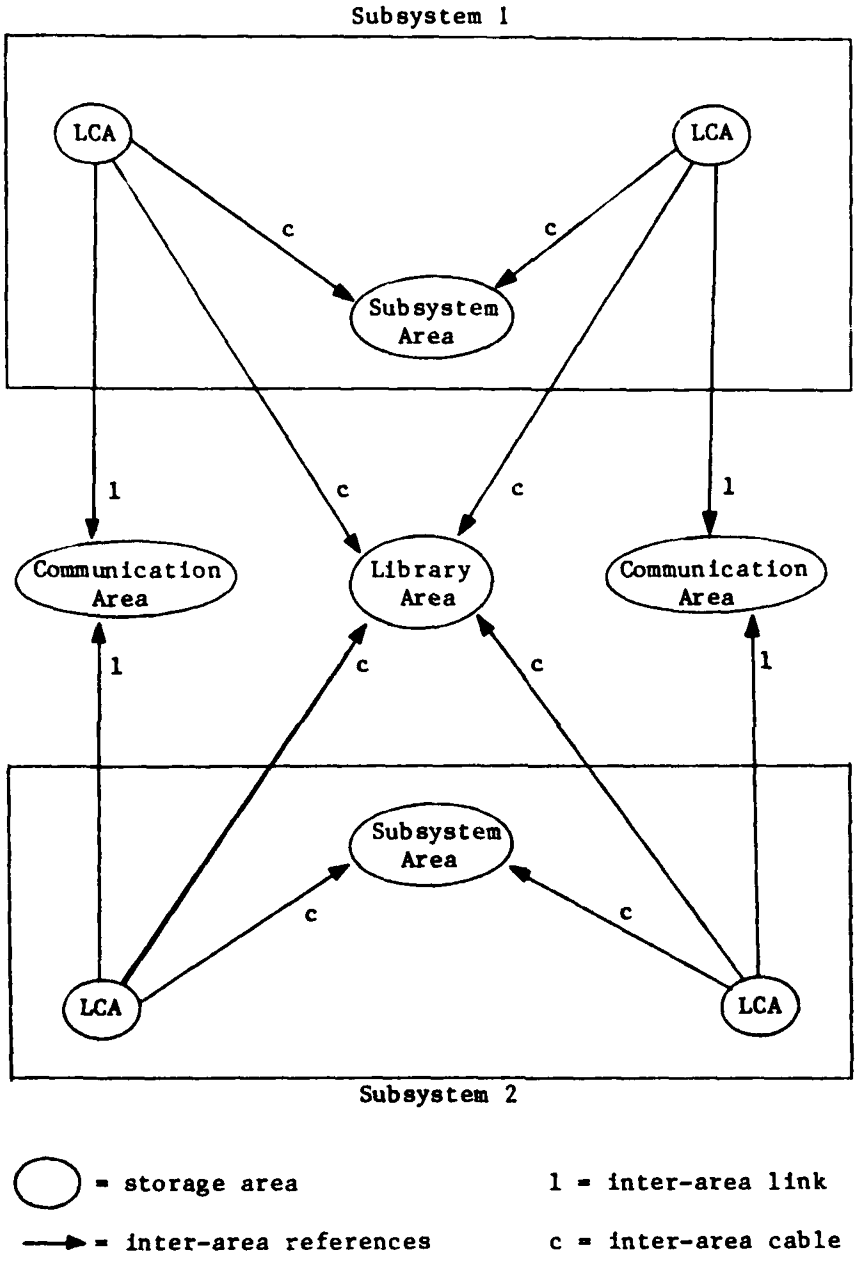


Figure 3.2. The subsystem model and storage areas.

these two categories. Less frequent inter-area references, for instance from one LCA to another, are handled by inter-area links.

AESOP does not recognize the existence of subsystems. Instead, the user will tell the AESOP implementation when a cable should be created from area A to area B through the storage_area$close(A, B) operation. By creating the correct cables, the user will, in effect, identify subsystems to AESOP and thus aid efficient execution.

3.4 Garbage Collection in AESOP

The subsystem pattern of use means that Bishop's basic mechanism for garbage collection is practical since inter-area links, which are neither time nor space efficient, are used infrequently. Cables, when used, are used in a manner that does not adversely effect the efficiency of garbage collection since the most frequently garbage collected areas, LCA's, do not have incoming cables so that most garbage collections only have to deal with one area at a time.

Bishop's garbage collection scheme could be used with no changes. However, his scheme results in processes being periodically delayed while a garbage collection occurs; for many applications, this may be unacceptable. This thesis proposes integrating Baker's real time garbage collector[3] into Bishop's scheme. Such a garbage collector allows AESOP to provide real time response to a user programs in that they will seldom have to wait for a garbage collection to complete.

3.4.1 Baker's Real-Time Garbage Collector

Baker's algorithm is a copying garbage collector that copies all accessible objects from a source space, called FROM space, to a virgin area of the same size called TO space. The garbage collector first copies all objects known to be accessible (i.e. the roots of the graph of accessible objects) from FROM space to the lower part of TO space. The garbage collector then successively examines each object in TO space and moves every object that it refers to from FROM space to TO space. At the same time the reference in TO space is changed to refer to the object's new location in TO space. When all objects in TO space have been examined all objects remaining in FROM space can be deleted since there are no valid references to those objects. The effect is that the garbage in FROM space is collected into one large free area in TO space since all referenceable objects will be at one end of TO space.

Baker's algorithm is "real-time" since it is interleaved with the normal execution of the user's program. Whenever the user attempts to create a new object (a LISP cons cell in his case), the new object is placed at the upper end of TO space and a few cycles of the garbage collector are run. In this way garbage is constantly being recovered and the system is unlikely to ever have to pause in order to allow a garbage collection to finish, thus the term real-time garbage collector.

3.4.2 Real-Time Garbage Collection in AESOP

Garbage collection in AESOP will be a combination of Baker's scheme, to get real-time response, and Bishop's ideas coupled with the subsystem model to increase the efficiency of garbage collection.

3.4.2.1 TO Space

Whenever an area is to be garbage collected the system must first find TO space. There are two choices for TO space. One possibility is to preallocate TO space for each storage area, thus doubling the amount of storage allocated to a storage area.[1] This has the disadvantage that storage areas have at most 50% storage utilization. Alternatively, TO space could allocated at the time that the decision is made to garbage collect a storage area. This has the disadvantage that a new storage area must be created at the beginning of each garbage collection. A mixed approach seems appropriate for AESOP since storage areas in AESOP fall into two basic categories: small, very active areas (e.g. LCAs) and large, slowly changing areas (e.g. library areas).

The very active areas tend to rapidly generate garbage so they need to be constantly garbage collected. TO space should be pre-allocated for these areas for two reasons. First, the delay and overhead involved in creating TO space, even if small, is avoided.

[1] This is the policy proposed by Baker. Also note that in this case the roles of the two parts vis a vis TO and FROM space change after each complete garbage collection.

Second, the process can always perform a garbage collection - it never dies because TO space could not be created.

For the less active areas, which also tend to be very large, TO space should be created when the area is about to be garbage collected. In this way the storage overhead associated with those areas is relatively small (and certainly results in more than the 50% utilization possible when TO space always exists). The danger with this scheme is that there might not be enough free physical storage to allow TO space to be allocated when the garbage collection is initiated. Although a possibility, it will be ignored here and will be returned to when physical memory management is discussed in the next chapter.[1]

Handling TO space in this way is reasonable, but it is now necessary to classify storage areas into these two categories. This can either be done explicitly by the user or as a result of measurements performed by the implementation. The first requires a new operation on storage areas to allow declaring that a storage area should have a pre-allocated TO space. This is a case where a particular choice of implementation leads to a desire for new architectural operations. That is, the separation between implementation and architecture breaks down with this choice of implementation. The alternative is to have the implementation make a decision based upon the size of an area and its measured rate of garbage generation. The actual manner in which this

[1] In the event that it does occur, it is possible, at the expense of additional complexity in the implementation, to perform a classical mark/sweep garbage collection on that area to avoid the problem.

decision should be made is, however, a problem for further research. This thesis will take a third approach. Any storage area that is used as the default storage area for a process, either via a process$create or a process$set_default_area operation, will have TO space reallocated. This will cause garbage collection of the most active areas, LCA's, to be most efficient. All other areas will have TO space dynamically created.

3.4.3 The Roots of the Graph of Accessible Objects

To garbage collect an area it is necessary to find the graph of objects accessible in that area. This graph has its roots in four places. First the incoming inter-area links to that area name some of the objects that are accessible from other areas.[1] These objects are easy to find since each area has a list of all of its incoming links.

The second source is a system maintained list of all of the processes in the system. If a process is potentially runnable (i.e. not in the stopped state), the system must retain a reference to that process so that it can be executed since there is no requirement in AESOP that user programs retain such a reference. This reference

[1] Of course, due to the asynchronous nature of garbage collection in storage areas it is possible that at the time that one storage area examines its incoming links, the area that claims to need the referenced object may, in fact, no longer need it. Unfortunately, the area being garbage collected can not know this and must assume that all objects referenced by incoming links are needed.

ensures that all of the processes and all of the objects that they reference will not be garbage collected.

The third source is the distinguished object ROOT. It is defined to always be accessible so that all objects referenced by it are also accessible.

The fourth, and final, source of roots are the objects in storage areas cabled to the area being garbage collected. To determine which objects are accessible from a cabled area it is necessary to know which objects are accessible in that area and then trace accessible objects from those objects. In his thesis Bishop acquired this information by garbage collecting all of the areas transitively cabled to the area that needed the garbage collection at the same time as that area was garbage collected (i.e. he treated this set of areas as one large area for garbage collection purposes). Here a slightly different scheme is proposed. Whenever a garbage collection is initiated in a storage area, call it A, each incoming cable to A will have a flag gc-in-progress associated with it that is set to true at this time. This flag informs the areas cabled to A that, while garbage collecting, they must "mark" all objects in A that they directly refer to by moving those objects from FROM space to TO space. Whenever the cabled area completes its garbage collection it sets the gc-in-progress flag to false. This informs the garbage collector of A that all objects directly accessible from the cabled area have been moved to TO space. Thus when all gc-in-progress flags are turned off in incoming cables to A, all the roots of the graph of accessible objects in A have been marked so that A's garbage collection may finish.

This technique requires that there be no cycles of storage areas. Suppose area A was cabled to B which was cabled to A. Now suppose a garbage collection of B is initiated. It can not finish until A is garbage collected, but A can not finish until B is done. Obviously, the cycle never ends. The storage area type manager is responsible for ensuring that such cycles do not occur. Moreover, note that if the subsystem model of execution is followed, no attempt will ever be made to create a cycle.

The resulting garbage collector is space efficient. The working set of the garbage collector when garbage collecting A is about the size of A plus the very end of TO space for each area that A is cabled to (if it is currently being garbage collected) plus the very end of A's own TO space. Contrast this to Bishop's scheme in which the working set of the garbage collector is the combined size of all of the areas transitively cabled to the area being garbage collected.

3.4.4 The Garbage Collection Algorithm

This section describes the procedure for garbage collecting a storage area A. This algorithm does not contain any synchronization to handle parallelism because the parallelism present is only pseudo-parallelism. This issue will be returned to below.

First TO space is created as previously described. All incoming cables to A now have their gc-in-progress flag set to true. For each outgoing cable from A, associate a flag was_gcing that is now set to true only if the area that A is cabled to is currently being garbage collected otherwise it is set to false. This flag will be used by A to correctly set the gc-in-progress flag in this cable when the garbage collection of A finishes.

The garbage collector of A now moves all objects accessible through inter-area links and from ROOT to the lower part of TO space. These, in conjunction with the objects being moved by cabled areas being garbage collected in parallel, define the roots of the graph of accessible objects. The garbage collector now sequentially examines every object in the lower part of TO space. For each such object it examines all of the pointers, call one of them P, in that object.

If P refers to an object in FROM space then that object is moved to the lower part TO space. The copy in FROM space is marked as "forwarded" and its address in TO space stored in the old copy. Finally the pointer P is changed to refer to the object's location in TO space. If P refers to an object that has been forwarded, P is updated to refer to its new location. If P refers directly to an object in TO space, nothing need be done. If P refers directly (i.e. not through a link) to an object that is in another storage area, call it B, and if B is being garbage collected, P is handled as above except that TO/FROM space are B's and not A's. If B is not being garbage collected then nothing need be done. If P refers to a link in FROM space, a copy of that link is created in TO space and P modified to refer to it.[1]

The garbage collector continues examining objects in TO space until all gc-in-progress flags in all incoming cables to A have been set to *false* so as to ensure that parallel garbage collections have marked all accessible objects in A. All outgoing links from FROM space are then removed from other area's lists of incoming links since they are no longer needed by A. Once this is done, all of the objects *remaining in* FROM space are known to be inaccessible so FROM space may be destroyed. This collects the storage consumed by unused objects, including inaccessible links, into one free area at the end of the upper end of TO space

At this point, for each outgoing cable C from A to B, the garbage collector sets C.gc-in-progress to C.gc_in_progress *and* *not* C.was_gcing. This ensures that C.gc_in_progress is set to *false* only if all objects in B that are directly referred to by objects in A have been marked. In particular, if B began a garbage collection after A began its garbage collection, there may be objects in B that are accessible from A but have not been marked (recall that A will only mark objects if the target

[1] The garbage collector can cause references from A to share links by checking for the existence of this link in TO space before creating a new one and using the old one if it exists. This results in greater storage utility in A at the expense of additional complexity in the implementation.

area is beging garbage collected). The algorithm is summarized in Figure 3.3 and given in more detail in Appendix B.

The algorithm is pseudo-parallel in that one or more cycles of the garbage collector, a cycle being an execution of steps 7a and 7b, occur on each reference to A. However, a step of no other garbage collector can run while a step of this garbage collector is running so long as the implementation prevents interrupts from occurring (recall that a uni-processor implementation of AESOP has been assumed). The pseudo-parallelism occurs since after a few cycles of the garbage collector are run, references by user′s programs are executed, which may result in running a few cycles of the garbage collector in another area.

3.4.5 Initiation of Garbage Collection

This algorithm does not specify when to initiate the garbage collection of a storage area. Garbage collection of an area A will be initiated in five ways.

First, a garbage collection will be initiated when an attempt to create an object fails due to lack of storage. This is an undesirable technique if it is the only one used since processes may experience long delays if a creation triggers a garbage collection.

Second, as proposed by Bishop, the system can measure the creation/deletion activity in A and initiate a garbage collection when a threshold is reached. A properly chosen threshold value will minimize

The procedure to garbage collect storage area A.

1. If area A does not have a TO space associated with it, create one of the same size as A.

2. Set all gc_in_progress flags in incoming cables to A to true.

3. For each outgoing cable C from A, set C.was_gcing to C.gc_in_progress.

4. Move all objects referred to by incoming links to TO space, leaving a forwarding pointer to the object's new location in its old location in FROM space.

5. Repeatedly perform step 6 until all incoming cables have their gc_in_progress flags set to false and all objects in lower TO space have been examined.

6. Examine the next object in lower TO space and for each pointer, P, in it perform steps 7a and 7b.

7a. If P refers to an object, including a link, in A, ensure that the object is in TO space, leaving behind a forwarding pointer in FROM space, and update P to refer to the object's new location.

7b. If P directly refers to an object in another area and if that area is being garbage collected, perform step 7a using that area's FROM and TO space.

8. Remove all links in FROM space from the lists of incoming links to areas.

9. FROM space now contains only free storage and may be destroyed.

10. For all outgoing cables C from A, set C.gc_in_progress to C.gc_in_progress and not C.was_gcing.

Figure 3.3. The AESOP garbage collector.

unnecessary garbage collection activity while also eliminating delays to programs due to garbage collections.

Third, a garbage collection in all areas cabled to A will be initiated whenever a garbage collection of A is begun. This causes the garbage collection of A to finish as soon as possible and, to the extent that the subsystem model of use is true, causes areas of maximum garbage to be quickly garbage collected since areas with outgoing cables are more active.

Fourth, the implementation can spontaneously initiate garbage collections on storage areas when AESOP is likely to be idle for a while. This allows garbage collection to impact the user of AESOP in a minimal fashion. To provide this feature AESOP needs a new interface to permit the user to say "I will not be needing the machine for a few hours". Such a change to AESOP is easily made. It is yet another example of a particular implementation of AESOP desiring specific architectural features.

A fifth possibility is to garbage collect all areas at all times as proposed by Baker. This has the advantage that AESOP will never pause to complete a garbage collection. It is rejected as the general policy since it results in unacceptably low storage utilization (at most 50% due to the presence of a TO space for all storage areas at all times). Instead, only those areas that have an integral TO space (i.e. LCA's) will be constantly garbage collected. Since these areas generate garbage rapidly, this should keep processes from having to pause for a garbage collection on their LCA to occur.

3.5 Object Creation and Deletion

Running programs will request the creation and deletion of objects. The program must specify the storage area in which a newly created object is to be placed. When an object is deleted, its storage is returned to the storage area the object belonged to. This section discusses the implementation of object creation and deletion.

3.5.1 Object Deletion

Objects are deleted and their storage reclaimed by one of two means: the garbage collector may determine that the object is no longer referenceable and thus its storage is no longer needed or the user may perform a delete operation on an object. The garbage collector case is easy. When the garbage collector determines that an object is no longer accessible, all of the storage used by that object is returned to the free storage pool implicitly during the garbage collection process when FROM space is destroyed.

AESOP allows the user to explicitly delete individual objects and to delete objects implicitly by deleting all of the objects in a given storage area. When an object is deleted there will, in general, still be outstanding references to that object. The system is responsible for invalidating those references. Moreover, the implementation should make an effort to utilize storage freed as a result of these operations available as soon as possible since one of the prime motivations for including delete operations in AESOP was to regain the use of storage quickly.

When an object is deleted in an area A, the implementation will mark all of the storage associated with that object as free except for its first word. This is done by first incrementing a counter of the amount of free storage in A, total_free_words, by the amount of freed storage and then placing the freed storage on a list of free storage that A maintains. The first word of the object is be marked as "deleted" and any further attempts to reference the now deleted object will result in the raising of the unexpected_deleted_operand exception. Thus the first word serves as a tombstone for the deleted object. The storage occupied by the tombstone is later reclaimed by the garbage collector for A. Whenever the garbage collector notices a reference to a tombstone, that reference is changed to say "deleted". Thus after a garbage collection is complete the storage occupied by the tombstone is reusable since there will be no outstanding references to it.

3.5.2 Object Creation

When an object is created some storage must be allocated to hold that object. The AESOP implementation is responsible for allocating that object within a storage area; the user is responsible for choosing which storage area it should be allocated in. This occurs since AESOP has taken the point of view that users may be interested about storage allocation in the large (i.e. deciding which objects belong in which storage areas) but they have no desire to participate in storage management in the small (i.e. the placement of objects within a storage area).

Numerous storage allocation strategies have been proposed in the literature and their performance analyzed. The strategy chosen for the implementation of AESOP must satisfy two criterion. First, the CPU time required to perform an allocation must be small. If allocations were to take a long time to perform, system performance would suffer due to their frequent occurrence. Second, the allocation strategy must not use large amounts of storage for bookkeeping purposes. Since there will be large numbers of potentially small objects, a large overhead will degrade memory utilization to an unacceptable extent. Any algorithm that meets these two goals will do for an allocation strategy.

At any given time, the storage in an area A looks like that in Figure 3.4. The partially allocated area contains both allocated objects and, on the free list, the storage returned by the explicit

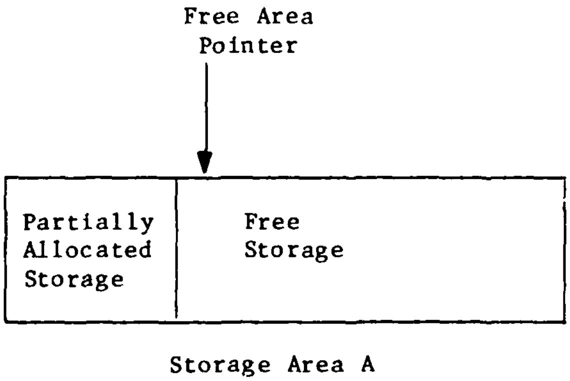
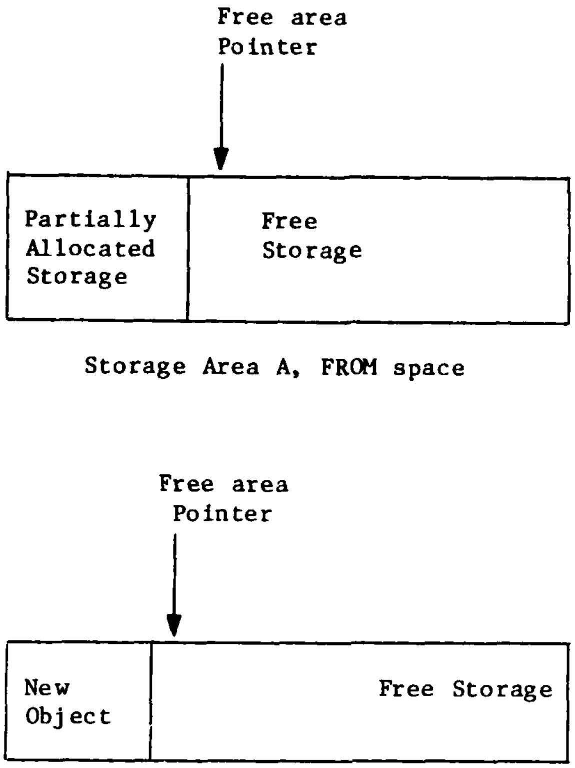


Figure 3.4. The storage within a storage area.

deletion of objects. Allocation of objects can take place either in the free area or by taking space from the free list. To allocate an object in the free area it is only necessary to remove an appropriate amount of storage from the free area by moving the free area pointer. To allocate storage from the free list, that list is searched for a suitably sized free area which is removed from the list. The object is then created in that area and any excess storage returned to the free list.

If the free area is not large enough to contain the new object, and if no suitably sized area is found on the free list, then, in general, a garbage collection must be performed on A to collect all of the free area in A into one contiguous area so that the allocation may be retried. If total_free_words is at least as large as the size of the object to be created, a new garbage collection cycle is initiated, the new object allocated at the end of TO space and total_free_words decremented by the size of the new object. The situation after this allocation will be as in Figure 3.5. This is guaranteed to work since the garbage collection will produce a compacted free area at least of size total_free_words at the end of TO space. It may, in fact, be larger due to collected garbage. If, on the other hand, the size of the new object would be greater than total_free_words, the allocation request can not yet be granted as there is no guarantee that sufficient free storage exists in A to satisfy the request. There may, though, be sufficient free area to satisfy the request once the area is garbage collected. The implementation at this point has two choices. It can refuse the allocation by signalling an error, say



Storage Area A, FROM space

Storage area A, TO space

Figure 3.5. A storage area being garbage collected

insufficient_space_now, in which case the program may try again later. Alternatively, the implementation may wait until the next garbage collection cycle terminates and try the allocation then. The allocation may fail then since, due to the use of storage areas by parallel processes, even if there is sufficient free storage (including the garbage of the area) when the garbage collection is initiated, there is no guarantee that it will be there when the garbage collection terminates since those parallel processes may create objects and thus use some of the free storage. This last occurrence will be rare since it will only occur when an area is very full.[1] This implementation makes the second choice since it succeeds in most cases. If, just after the garbage collection terminates, the allocation can not be performed, an error is returned to the requestor.

3.6 Multi-Area Cycles of Garbage

The garbage collection algorithm just presented does not discover all of the garbage in the system even if all programs in AESOP are stopped and an infinite number of garbage collections performed. To see this, consider Figure 3.6 in which the only accessible objects are ROOT, A and B. However a garbage collection of S1 will find that A and C are both accessible while a garbage collection of S2 will find that B and D are accessible. Thus neither C nor D will ever be considered garbage and thus have their storage reclaimed. In a long running system such

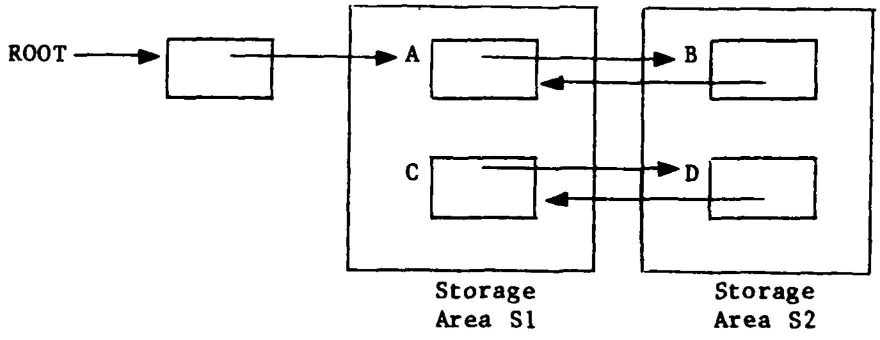


Figure 3.6. An example of a multi-area cycle of garbage.

[1] Since full areas will tend to be constantly garbage collected, their presence will degrade system performance. Since users want good performance, nearly full storage areas should occur only rarely.

reclamation is vital. Since the garbage collection algorithm of this thesis is based upon Bishop's work, it is reasonable to ask how he solved the problem. Bishop solved it by associating a directory with every storage area. The directories, as a group, are the roots of the graph of accessible objects in the system and can be used, as shown by Bishop, to eliminate inter-area cycles of garbage in a simple and elegant manner. This scheme is not applicable to AESOP since there are no directories provided by AESOP.

An algorithm to reclaim multi-area cycles of garbage must take into account how frequently such cycles occur in order to limit, if possible, 'he complexity of the algorithm. This thesis assumes that long term inter-area cycles of garbage are rare. This comes from the following consideration: storage areas are generally either temporary or are long term. An inter-area cycle of garbage involving a temporary area is unimportant since the cycle will be broken when the temporary area is destroyed. Objects in long term storage are probably part of a file system and cycles in file systems probably only occur as a result of entries in directories and such loops will be broken when the directory entry is deleted. The remaining inter-area cycles, which are the ones of interest, should be rare so that the detection of inter-area cycles of garbage need not occur rapidly. Since they are rare, it is inappropriate to devote large amounts of resources to discovering them. Thus a scheme with low overhead is desired.

Inter-area cycles of garbage in AESOP will be detected by performing a mark/sweep garbage collection of the entire system. The system-wide mark phase may take a long time to complete since it requires all storage areas to be individually garbage collected at least once. Since storage areas are garbage collected at varying rates, this may take a long time. However, due to the assumed rarity of cycles, this is not be a problem. The sweep phase occurs by making one, linear pass over all of storage, reclaiming all unmarked objects.

The algorithm starts at ROOT and marks all objects accessible from ROOT. It then recurses through the logical memory until all objects have been marked at which point unmarked objects will be reclaimed. This algorithm is unique in that no stack is used, nothing is done to make the system unusable while the garbage collection is in progress and it employs no centralized control mechanism. Instead, a distributed control algorithm, distributed in the storage areas, is used in which each storage area keeps two additional bits of information as state.

Associate with every object a single bit, its mark bit, that tells whether the object is accessible from ROOT. With every storage area associate two bits, ITID (for I Think I'm Done) and rescan. An area turns on its ITID flag when it thinks that all accessible objects in it have had their mark bit turned on. The rescan flag is turned on by other areas to indicate that the area needs to be rescanned for accessible objects before its ITID flag can be turned on. In addition, there are two global (system wide) flags: global_gc that indicates that a mark/sweep garbage collection for inter-area cycles is underway and gc_sweep that indicates when the sweep phase of the mark/sweep garbage collection is underway. What follows is a high level discussion of the basics of the algorithm. Appendix B should be consulted for complete details.

While an area is being garbage collected, whenever an object is moved to TO space its mark bit is turned on if it was already on or if it was referred to from an object whose mark bit was on. This results in the mark bits propagating through the graph of accessible objects. The ITID flag is turned on by an area whenever it finishes its garbage collection so long as its rescan flag is off. The rescan flag is turned on for an area A and its ITID flag turned off if a previously unmarked object in TO space is marked and the garbage collector for A has a⊥ready scanned that object and traced its references. This ensures that all objects in an area are marked correctly before terminating the mark phase. The mark phase ends when all ITID flags are turned on. At this point the global_sweep flag is turned on to allow all unmarked objects to be reclaimed and their storage added, in its entirety, to it's area's free list (i.e. a tombstone is not left behind as there are known to be no valid references left to this object). This sweep can occur without stopping AESOP since it only touches inaccessible objects. The following two points are also important:

1. A newly created object O should have O.mark set to the value of global_gc at the time that the object is created.

2. When garbage collecting an area, if global_sweep is true, pointers from objects with their mark flag off should not be followed.

The first ensures that objects created after the initiation of a global mark/sweep do not have their storage accidently, and incorrectly, reclaimed. The second speeds up the reclamation of storage since objects known to be inaccessible from ROOT (i.e. objects that do not have their mark bit on) are not traced from. This is the basics of the algorithm. Appendix B contains the complete algorithm and integrates it into the basic AESOP garbage collector. It also contains an argument as to why it terminates correctly.

This algorithm adds some complexity to the garbage collection algorithm used by each storage area in order to handle mark bits but this is a small burden, especially when implemented in hardware, since it just involves setting some bits in objects and in storage areas.

3.7 A Stack Mechanism for Local Name Spaces and Control

Control flow within a process follows a strict stack discipline (i.e. the call/return paradigm) indicating that a stack implementation of control is optimal. The LNS for a procedure activation is, in essence, the stack frame for that procedure so a stack mechanism for local name spaces would also seem to be ideal. Integrating stack allocation of local name spaces in the stack oriented control flow would be the best of all possible worlds. Unfortunately, an LNS is an object and, as such, arbitrary references to an LNS may be stored within AESOP. Such references are used, for instance, in creating closures to meet various language needs as discussed in chapter five. The ability to freely copy references makes a stack oriented deallocation scheme potentially unsafe since, when the stack algorithm says to deallocate the LNS, there may still be outstanding references to the LNS. Leaving behind a tombstone in the stack is unacceptable as it would prevent the stack from easily growing again. However, this section shows how the stack allocation of local name spaces can be made to work correctly and efficiently.

When an AESOP process is created, a default storage area is specified. That storage area will be used to hold the stack for the newly created process by reserving a portion of it for the stack and using the remainder for allocating objects. When a procedure is called, an activation record is pushed onto the stack followed by the newly created LNS. The activation record contains the address of the calling instruction (a code segment and an offset within that code segment), a back pointer to the previous activation record and a specification of the GNS of the caller (to permit restoring the callers environment upon return). The activation record is followed by a pointer (initially null) to an LNS tombstone (to be described later), and, finally, the LNS. Figure 7 shows the possible configurations.

In deallocating an LNS in a stack manner it is necessary to ensure that no references to the LNS remain after it is deallocated. To do this in an optimal manner is probably hopeless; instead the common cases will be covered here. This algorithm notes that the initial situation, in which an LNS refers to itself as a result of the procedure call mechanism, is safe and then notes some safe transitions from safe states.

Code Segment	Instruction Offset	Back Pointer	GNS Spec.	Ref. to an LNS Tombstone	LNS

Basic format of an activation record

Code Segment	Instruction Offset	Back Pointer	Null Ref.	Ref. to an LNS Tombstone	LNS

The result of proc$call or a type manager call.
(the caller's GNS is the same as the called procedure's)

Code Segment	Instruction Offset	Back Pointer	GNS ref.	Ref. to an LNS Tombstone	LNS

The result of a proc$call_with_gns call.

Code Segment	Instruction Offset	Back Pointer	GNS ref.	LNS ref.

The result of a closure$run operation

Figure 3.7. Formats of an activation record.

If a reference to an LNS is copied within that LNS, no problem results since the new reference disappears when the LNS is deallocated. A reference to an LNS may also be placed into a closure. This is safe so long as the closure is only known in that LNS.

The LNS (or a closure referencing it) may also be passed as a parameter. This is safe so long as the reference to the original LNS does not leave the newly created LNS. Applying this argument recursively, it can be seen that so long as references to an LNS stay higher in the stack[1] than that LNS, no problems result. Also, if a reference to an LNS should get into a closure then everything is safe so long as that closure is only known higher in the stack than the LNS it refers to. It is claimed, based upon the examples of using AESOP that are presented later in this thesis and upon consideration of the ways in which languages generally need to treat naming environments, that the vast majority of all references to local name spaces will fall into one of these categories. Thus an implementation that traces these safe transitions will greatly aid performance.

In consideration of the arguments of the last paragraph, every object reference contains a stack reference flag that, when on, indicates that this object reference is to an entity associated with the stack allocation of local name spaces and so should be treated carefully. Whenever an attempt is made to copy such a reference other than within an LNS or as part of the procedure call mechanism, the algorithm in Figure 3.8 should be executed. The result might be as in Figure 3.9.

[1] A reference is higher in the stack if the LNS it resides in was created after the LNS that it refers to. That is, its procedure activation is more recent.

1. Let OR be the object reference in question and O the target of the copy operation.

2. If O is an LNS and OR refers to an LNS (either directly or indirectly through a closure) no higher than O in the stack then return as no further action is needed.

3. If O is a closure that is being created then create it, set the stack_ref flag on in the reference returned by closure$create and return.

4. At this point, OR is moving to a place which may result in a dangling reference when the referenced LNS is destroyed. Thus drastic action is called for.

5. Create an "LNS tombstone" for the LNS referred to by OR off of the stack, if not already created. Place a forwarding pointer to it in the LNS tombstone reference field in the activation record corresponding to the referenced LNS. Mark the tombstone as "forwarded" and place a reference to the LNS in it.

6. If OR refers to an LNS, update it to refer to the LNS tombstone for that LNS.

7. If OR refers to a closure, update the LNS reference in the closure to refer to the LNS tombstone and set OR's stack_ref flag off.

Figure 3.8. The algorithm for handling the stack of LNS's.

A number of remarks about this algorithm are in order. This algorithm looks for the movement of references to local name spaces to places where a stack oriented deallocation for those local name spaces might result in errors. At that point, a forwarding pointer for the LNS in question is made out of the stack. The reference leaving the stack then refers to this tombstone and will be forwarded to the correct LNS when used. When the procedure corresponding to that LNS returns, the

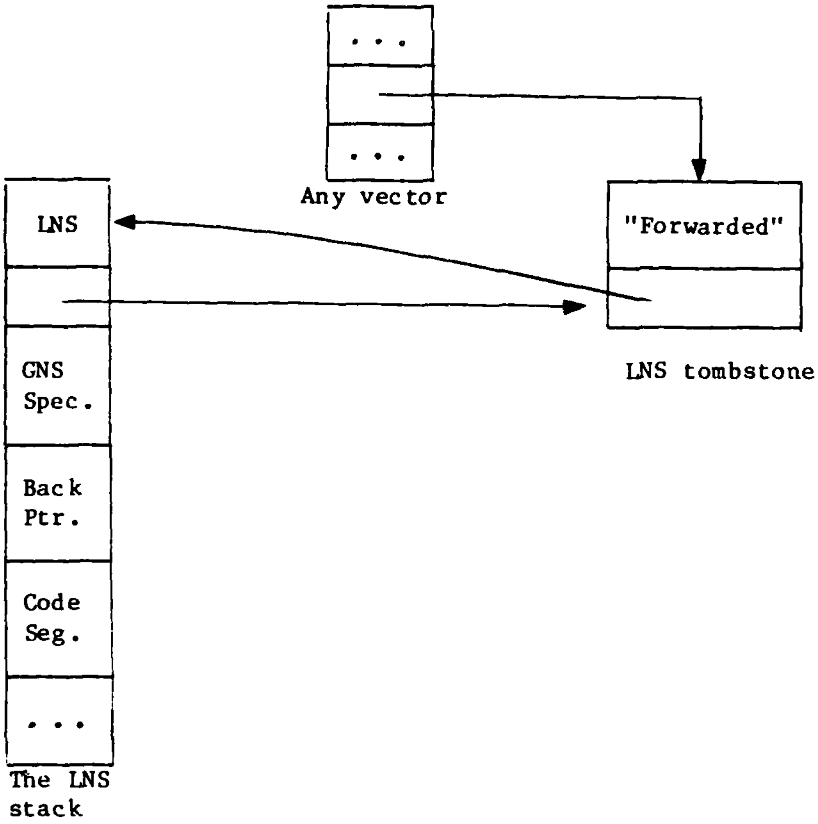


Figure 3.9. An LNS reference moved from the stack

tombstone is found and marked "deleted" so that all remaining references to that LNS will fail. Meanwhile, the stack area is uncluttered by the tombstones that would adversely effect a stack allocation strategy. Also, tombstones for an LNS are only created when needed so that the remainder of the area containing the stack remains uncluttered from this source.

It should be noted that this algorithm is suboptimal. Consider Figure 3.10 and suppose that it depicts all references in AESOP. As it stands LNS1 can be safely deallocated in a stack manner. Now suppose that the procedure executing in LNS2 does V(P) := LNS2(1) and then returns. The above algorithm will cause a tombstone to be created for LNS1 off of the stack even though that is unnecessary since V is inaccessible once LNS2 is returned from. Thus the algorithm is not optimal.

There is one remaining question concerning this algorithm - how can the creation of a tombstone in step 6 be guaranteed to always be possible? Could it not occur that there is insufficient free storage in the storage area containing the stack to permit the copy? When an activation record is pushed onto the stack, reserve enough free storage to create an LNS tombstone by decrementing total_free_words for the storage area containing the stack. Now when an attempt is made to

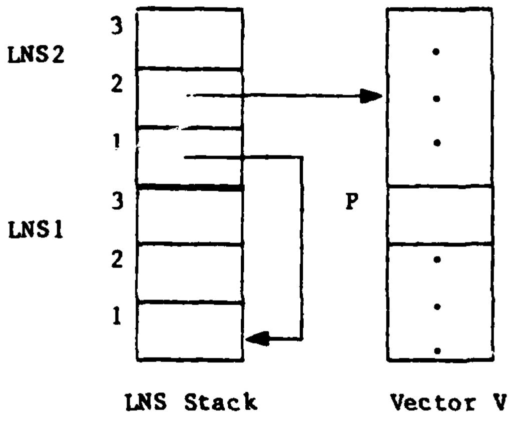


Figure 3.10. An example of suboptimality.

create the tombstone for the LNS, it must succeed, perhaps after a garbage collection, since storage has been reserved for it. When the procedure returns, if no tombstone has been created, the storage reserved is returned to the free pool by incrementing total_free_words.

3.8 Conclusions

This chapter has described a logical storage management implementation that could be used in an implementation of AESOP. The management of the logical memory, the memory that contains the AESOP objects and the unused storage in AESOP, is done in terms of the storage area mechanism using the hypothesized access patterns of the subsystem model to improve performance.

Garbage collection proceeds on a per-area basis. Inter-area references that effect the garbage collection process are handled either by inter-area links, which have little effect on the garbage collection process, or by garbage collecting areas that are cabled to the area being garbage collected. This multiple garbage collection is not an impediment to efficient system operation since areas with outgoing cables tend to need to be more frequently garbage collected due to their higher rate of activity under the subsystem model. This mechanism is similar to that proposed by Bishop but differs in that a real-time garbage collector is incorporated.

The problem of multi-area cycles of garbage has been addressed by designing a mark/sweep garbage collection algorithm. This algorithm is new in that it employs no centralized control and occurs during normal system operation. It has been designed to cost little by being made part of the basic garbage collection mechanism within storage areas.

Finally, this chapter has shown how local name spaces can be created and destroyed in an efficient and safe manner. This means that the frequent use of procedures in AESOP is not an impediment to efficient operation.

Chapter Four

Other Issues in Implementing AESOP

Chapter three has given an overview of the issues involved in implementing AESOP and discussed the management of the logical memory defined by AESOP. This chapter discusses the remaining issues associated with implementing AESOP: managing the physical resources underlying the AESOP implementation (especially memory) and implementing AESOP's basic types.

This chapter first describes the hardware assumed to underly this implementation of AESOP. Next, the format of object references is discussed. Object references must be small so as to minimize space wastage within them since they make up every AESOP object while, at the same time, must be large enough to permit referring to any AESOP object. The problem of managing the physical memory underlying AESOP is discussed next. The approach taken here is to treat the actual AESOP memory as a paged, virtual memory. The problems associated with allocating physical storage to storage areas are discussed next. This implementation allocates contiguous blocks of secondary storage to storage areas so that the problem of memory fragmentation is of concern. Next, the implementation of AESOP's basic types is discussed. Most of the basic types are trivial. The io type manager is discussed at length since its implementation allows I/O devices to interface with an AESOP system. The storage area type manager is important since it embodies the logical storage algorithms. The process type manager is discussed to show that AESOP processes can be provided cheaply. Finally, some of the ways in which special hardware, such as associative memories, can improve the performance of an AESOP processor are discussed.

4.1 Fundamental Hardware Assumptions

This implementation of AESOP is based upon a single, central processor that manipulates a passive primary memory and a passive secondary memory. The memories are passive in that they only store data and do not provide other facilities.[1]

The processor is specially designed to implement the semantics of AESOP as defined in chapter two. The processor is responsible for managing both the multi-level physical memory and the logical memory defined by AESOP. In general, it is responsible for using the physical memory, a collection of uninterpreted bits, to create the object memory defined by AESOP and for providing the facilities for manipulating that memory.

Primary memory will consist of some quantity of 32 bit words. This word size been chosen to be large enough to meet the perceived addressing needs of AESOP while being small enough so that the overhead of one object reference stored per word is not onerous. The actual

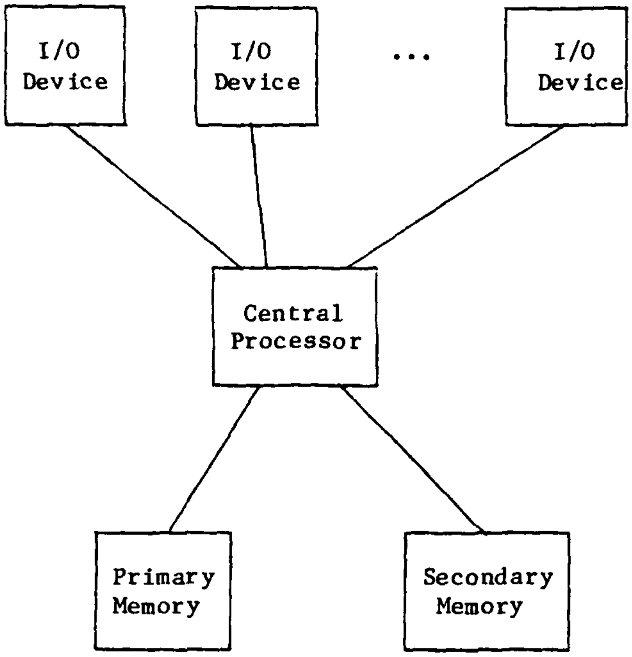
[1] Later in this chapter it will be seen how some simple relaxations of this assumption lead to more efficient implementations of AESOP.

quantity of primary memory will vary from machine to machine and will be chosen to provide adequate performance for a particular user.

Secondary memory will consist of one or more large capacity devices such as disks or bubble memories. The amount of secondary storage will vary from one machine to the next; the only assumption is that there is sufficient secondary memory to meet the user's needs. The most important characteristic of secondary memory is its relatively slow access time relative to primary memory access time (25 or more milliseconds for accesses to a disk versus a microsecond or less to access primary memory).

I/O devices are attached to AESOP through the central processor. The processor cooperates with those devices to meet the semantics defined in chapter two, i.e. to allow the device to manipulate event counts and bit vectors provided by the programs using those I/O devices and to allow programs to access and manipulate a device's status registers.

The entire system has the configuration shown in Figure 4.1. Alternative hardware configurations involving multiple processors are possible. However, such configurations increase the complexity of the AESOP implementation since there is now parallelism within the implementation. The configuration chosen here permits the critical issues of implementing AESOP to be examined without the need to consider the irrelevant issue of parallelism.



---- = a data path.

Figure 4.1. The basic hardware configuration of AESOP.

4.2 Object References

Almost all objects in AESOP consist of a sequence of references, called object reference, to other objects. As such, it is advantageous to have all object references physically represented with entities that are all the same size to allow efficient random access to the references within an object. The object references must be small in order to minimize memory wastage within them (e.g. a 64 bit reference to a boolean wastes 63 bits) but large enough to allow for a suitably large space of objects since every object reference must be large enough to be able to refer to any object within AESOP.

This implementation of AESOP assumes that a physical address space of 2^{29} object references (approximately 500 million object references) is a reasonable upper bound on the size of the logical memory needed in most uses of AESOP, given that it is a personal computer. Object references are implemented in 32 bit words since, as will be seen below, this allows references to some built-in objects to be efficiently encoded while also allowing references to any other object within this address space of 2^{29} object references.

Object references may refer to the various built-in objects in the system (the built-in type managers, integers, characters and booleans). Some of these objects are immutable and their state small enough to be encoded within an object reference itself so that space and time efficiency are increased. To do this, the high order bit of every object reference is reserved for a tag bit that indicates whether that object reference is to one of these special items or is to some other object.

If the object reference refers to one of these special items, the tag bit is turned off and the remaining 31 bits of the object reference are used to encode that special item as shown in Figure 4.2. In this way these objects are represented in the references to the objects so the objects themselves require no storage. Also, references to deleted objects are represented in the object reference itself allowing for the reclamation, as seen in chapter three, of the tombstones left behind deleted objects.

<"01"b, a 30 bit integer> a reference to the given integer

<"000"b, an 8 bit character> a reference to the given character

<"0010"b, a single bit> a reference to the given boolean

<"0011"b, data> for a reference to a built-in type manager or for special object references. The field data is interpreted as follows:

Value	Reference interpreted as:
0	a reference to the boolean type manager
1	a reference to the character type manager
2	a reference to the closure type manager
3	a reference to the code segment type manager
4	a reference to the event count type manager
5	a reference to the integer type manager
6	a reference to the io type manager
7	a reference to the null type manager
8	a reference to the object viewer type manager
9	a reference to the procedure type manager
10	a reference to the process type manager
11	a reference to the sequencer type manager
12	a reference to the storage area type manager
13	a reference to the type manager type manager
14	a reference to the vector type manager
15	a reference to a deleted object
16	a reference to nil

Figure 4.2. The format of special object references.

A reference to some other object is indicated if the tag bit is on. In this case the remaining 31 bits of the object reference are the 29 bit address of an object (i.e. a reference to the storage representing the object) and a two bit type_ref field. These objects require representation in memory for two reasons. First, their state is too large to be represented within a 32 bit word. Second, and more importantly, these objects tend to be mutable so that encoding their state in the object reference is hopelessly inefficient - changing the state of such an object would involve finding and updating all references to the object.

As will be seen in the next section the home of objects is on secondary memory, primary memory is just a cache for objects. The 29 bit address in the object reference must name this object by, in general, naming its home. There are two ways of doing this. First, this address could be the secondary storage address of the object's home. The second possibility is to have this 29 bit address be an index into some indirection table. This implementation of AESOP will use the first scheme to avoid the overhead and complexity involved with using an indirection table. The penalty for this is that the allocation of secondary memory becomes a little tricky, as will be seen later in this chapter.

The type_ref field in the object reference indicates whether the object reference is:

1. A simple reference into the storage area containing the reference.
2. A reference to an object associated with the stack allocation of LNS's in the storage area containing the reference.
3. A reference through an inter-area link to an object in another storage area.
4. A direct reference to an object in another storage area that is cabled to the storage area containing the reference.

This field allows for the correct handling of the stack allocation strategy for local name spaces by allowing the simple detection of references to objects associated with that strategy. This format also allows an object reference to indicate that it refers to an object outside of the referencing area, making it immediately known whether or not an inter-area link or cable must be checked for when copying an object reference. In addition, it permits the garbage collector to discriminate between references through links and direct references to cabled areas.

Note that this second format for object references does not allow for type information in the reference itself. Thus, since AESOP strongly types objects, it is necessary to place the type of the object with the object itself. To accomplish this, every object in AESOP will be prefixed by a header giving the type of the object. In addition, the header will hold the per-object mark bit needed in the global mark/sweep garbage collection of chapter three as well as any other object specific information, such as the length of vectors, that is needed.

This format of object references has the advantage of being relatively short (only 32 bits are required) while still allowing for the space efficient encoding of some built-in objects and providing for the efficient handling of inter-area references. The interpretation of these object references might be difficult in a software implementation of AESOP but is trivial in a hardware implementation. In either case, the good points of these object references outweigh any implementation difficulties.

4.3 Allocation of Physical Storage to Storage Areas

Storage must be allocated for storage areas whenever a new storage area is created or an old area requires creation of TO space as part of garbage collection. This storage must occupy contiguous segments of address space since the logical memory management algorithms of chapter three have assumed that consecutive words in a storage area have sequential addresses. This implementation will use the physical secondary memory address space as this address space. That is, storage areas will occupy contiguous segments of secondary storage.

An alternative to contiguous allocation is to allocate storage areas in a paged, virtual address space. This has been rejected for two reasons. First, this layer of mapping takes time to perform and can only adversely affect performance. Second, the mapping table consumes large amounts of valuable storage. For instance, if pages are 256 words long, the 2^{29} size address space requires about two million page table entries - a large overhead for a personal computer.

Since storage areas are full fledged objects in AESOP, the algorithms of chapter three are directly applicable to allocating/reclaiming the storage of storage areas if all of secondary storage is regarded as a single storage area with the other storage areas considered as the objects residing i.. that single large storage area. To apply them, though, requires dedicating half of physical storage to FROM space and the other half to TO space. This results in at most 50% utilization of secondary storage which is unacceptable. Inst.ad, classical dynamic storage allocation techniqes for variable sized blocks (e.g. first fit, best fit and buddy systems) will be used.

Dynamic storage allocation algorithms have the problem of external fragmentation - as storage areas are created, destroyed and moved (as a result of garbage collecting an area), the free storage in the system becomes fragmented into small pieces and may result in the situation where the total free storage in the system is large enough to satisfy an allocation request but no single free storage area is large enough to satisfy the request. This would stop the requesting process until the pattern of secondary memory usage permitted the allocation - potentially a long time. However, Knuth has discovered[24] that, in practice, this is not a problem. A storage allocator that is in equilibrium (i.e. the same number of bits are being created and destroyed each second) is expected to run forever so long as the allocated objects are no larger than 10% of memory size. Memory utilizations of 90% are possible in this case. As a result, so long as the user of AESOP does not create storage areas that are too large, it is hypothesized that the only complete solution to external fragmentation, dynamic storage compaction in which storage areas are moved so as to collect all of free storage into one contiguous area, is unneeded. Instead, if AESOP should ever need to perform compaction, this implementation of AESOP will stop executing all AESOP programs and perform a simple compaction of secondary storage by moving all storage areas to one end of secondary memory. This movement is easy to perform since it will be the only activity in AESOP at that time.[1] Due to its rareness, the fact that this approach stops the system is acceptable. It should be noted that this particular implementation of AESOP could be modified, at a considerable increase in complexity, to perform this compaction dynamically. This has not been done here due to its great complexity relative to its expected frequency of use.

A storage area may be destroyed, and its storage returned to the free pool of secondary storage, by a storage_area$destroy operation or it may be garbage collected when there are no references to objects within the area and no references to the area itself. A garbage collected area is destroyed by returning all of its storage to its

[1] This compaction might also be done when the user says that the system will not be used for a while. The ability to say this was proposed for AESOP in chapter three.

parent as a credit and returning the storage used by that area to the free pool.

There are two possible ways to implement the storage_area$destroy operation on an area A. First, all objects in A as well as A itself can be marked as deleted. As garbage collection occurs in other areas, all references to these will disappear. Eventually, they will all be garbage collected. This method is simple but does not return storage immediately. Alternatively, at the time that A is destroyed, all references to A or the objects within A can be found and marked as a reference to a deleted object by modifying all links to A and linearly scanring all areas cabled to A. After this, A's storage may be returned to the free pool as there no longer any references to it. This method results in fast reclamation of storage at the expense of some delay before the operation returns. The second method is chosen here because a primary purpose of the delete operation is the rapid reclamation of storage.

4.4 Physical Memory Management

Chapter three has discussed the management of AESOP's logical memory. The previous sections have defined object references to contain a 29 bit secondary storage address of objects and discussed the allocation of secondary memory resources. This section will discuss the manner in which the address in object references is used to find the actual object and how the primary memory of AESOP is used to give reasonable efficiency. In essence, this section discusses a virtual memory mechanism for the AESOP implementation.

Given the 29 bit secondary storage address of an object, the implementation must find the contents of the object. The simplest approach is to always go to the specified secondary storage location, retrieve the object and, if necessary, modify it and immediately write it back to secondary storage. Since secondary memory is slow, and there do not appear to be any technologies that will change this, this would result in a hopelessly slow implementation of AESOP. Instead this implementation will use primary memory to encache objects likely to be referenced in the near future. Thus the problem of migrating objects between primary and secondary memory arises. This implementation takes the traditional approach of encaching the most recently used objects in primary memory, i.e. no prediction of future references is attempted.

There are two approaches to managing the physical memory: an object based one and a page based one. An object based scheme would bring an object into primary memory whenever referred to. This has the advantage that only those objects known to be needed are brought into primary memory so that I/O traffic to secondary memory can be minimized and greater primary utilization is, at first glance, possible. It has the disadvantage that a large amount of bookkeeping is needed to keep track of which objects are actually in primary memory and where they are in primary memory. This approach has been investigated by Snyder[47]. In a page based scheme memory is divided into contiguous blocks called pages. Whenever an object is referred to the page of storage that contains that object is brought into primary memory. This has the advantage that if related objects are kept on the same physical page, then a reference to one of those objects will cause all of them to be brought into primary memory with one reference to secondary memory, thus reducing the amount of secondary storage I/O traffic. If the mapping from secondary storage address to page identifier, the name of the page referred to, is simple, the overhead to keep track of the primary memory location of objects is small. It has the disadvantage that unneeded objects may be brought into primary memory unless the objects within a page exhibit locality of reference.

This implementation will use a page based approach. This decision has been made since the storage area mechanism and the method of garbage collection combine to make locality of reference a likely occurrence. In particular, a storage area, under the subsystem model of use, tends to have related objects within it. The garbage collection mechanism places objects that are within a single storage area physically close to each other if they refer to each other since the garbage collection scheme performs a breadth-first traversal of the graph of accessible objects.

The degree of locality of reference depends upon two factors. First, the branching factor of objects is important since an object that references 100 other objects will not be close to the 100'th object as the other 99 will physically be between them. However, as reported by Snyder[47], branching tends to be small (3 in the programs that he measured) so this is not a significant factor. Second, as objects are moved to TO space as a result of areas cabled to the one being garbage collected and as new objects are created, the locality of reference being produced by the garbage collector of this area is reduced since these objects are placed in the wrong part of TO space from the point of view of locality of reference. So long as object creation is not too frequent and inter-area references occur less frequently than intra-area references, this effect should be small. Thus locality of reference within a storage area should be good.

The final aspect of this scheme is the specification of a page replacement algorithm. Any of the many algorithms that have been used in other virtual memory systems or presented in the literature are appropriate so long as the scheme is not too complex so as to be a burden on a personal computer.

To make the translation from secondary storage address to page identifier easy, fixed sized pages are used. Thus a page size must be chosen. If the page size is too small, the storage needed to remember the mapping from page identifier to primary memory address will be excessive. On the other hand, if the page size is too large, the advantages of locality of reference will be lost since too much primary memory will be consumed by objects that are not likely to be referred to soon. The proper choice of page size will depend upon the actual characteristics of AESOP systems. At this writing such information is unavailable and is a subject for future investigation. For concreteness, a page size of 256 words will be chosen. Figure 4.3 shows the form of a normal object reference that results from these decisions.

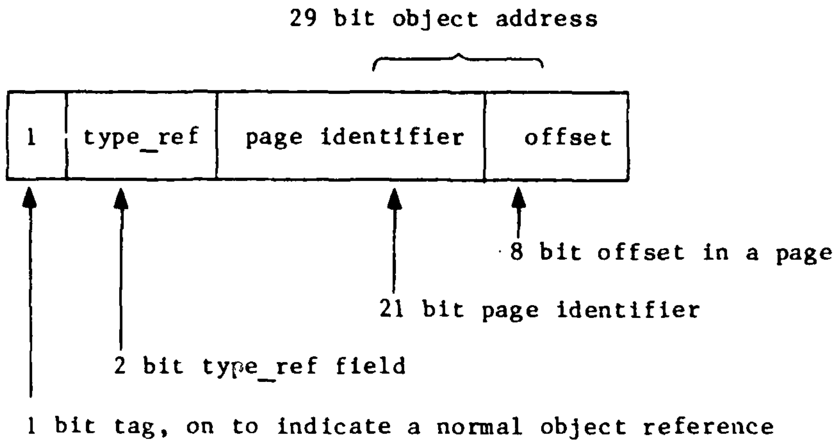


Figure 4.3. The format of a normal object reference.

Given a page identifier, derived from the secondary storage address of the object, it is necessary to determine its location in primary memory, if any. One possibility is a table that tells, for all page identifiers, where in primary memory the page is, if at all. However, this is nothing more than the secondary memory mapping table rejected earlier in this chapter for being too large so it is rejected here also. Instead, the system will maintain a table of all pages that are in primary memory that maps page identifiers into primary memory addresses, and supports the insertion, deletion and lookup of entries. This table is not too large since it need only map pages that are in primary memory.

There are numerous ways of implementing this table such as ordered lists, b-trees and hash tables. The actual mechanism must provide all three operations above in a speedy manner, with minimal storage overhead and without requiring a great deal of implementation complexity. An additional factor to consider is the parallelism possible if primary memory is divided into two or more modules, each of which contains one or more pages, and each module maintains a table of the pages within it. In this case the searching of these multiple tables can occur in parallel. In consideration of these factors, this implementation performs the page lookup function by a hash table since a hash table is easily implemented. Since this algorithm is simple, the option of placing the intelligence to manage it in every memory module is available to permit the parallel searching of these tables. There is nothing sacred in this choice; its important points are possible parallelism in memory modules and the simplicity in every such module. Another algorithm meeting these two points is equally acceptable.

Given the secondary storage address of an object (i.e. the address in an object reference), it is often necessary to ask the question "Which storage area is this object in?" This question must be asked, for instance, whenever an attempt is made to copy an object reference from one storage area to another. Answering this question efficiently is an important consideration in implementing AESOP.

There are three possible ways to answer this question. First each page of memory could be marked with an object reference to the storage area to which it belongs.[1] This has the disadvantage that answering

[1] For the time being the possibility of pages that are part of multiple storage areas is ignored.

this question requires that the page referred to must be brought into primary memory. Alternatively the system could maintain a table that contains, for every page in the system, an object reference to the storage area that it belongs to. As seen in the previous section, this requires a large table to be kept by the system. The third possibility is to maintain a table mapping ranges of page identifiers into storage area references. This is just a space optimization of the preceding scheme that is achieved by compressing the table mapping page identifiers into storage area references.

If usage follows the subsystem model the number of storage areas in AESOP will be relatively small - a few subsystems with a few processes in each, a few library areas and a few inter-subsystem communication areas. This will result, most likely, in only a few hundred storage areas in AESOP at any time. As a result, since storage areas occupy contiguous sequences of pages, the compacted table of the third scheme above is chosen in this implementation of AESOP. There are many ways to arrange this table, such as b-trees and ordered lists, and the possibilities are well documented in the literature. For concreteness, this implementation will use b-trees since they provide bounded lookup times while still being easy to manage. Again, the reader may feel free to substitute an alternative choice that meets the major criterion of implementation simplicity.

The problem of pages that belong to multiple storage areas will now be addressed. The architecture as presented in chapter two allows the creation of storage areas of arbitrary size so that they might not use all of a page. As a result, the possibility of a page being used by multiple storage areas exists. Although an algorithm for handling pages that belong to multiple storage areas can be developed, the complexity it adds to the AESOP implementation is non-trivial. This implementation of AESOP avoids this problem by always allocating storage areas so that they occupy an integral number of pages. User defined quotas will be rounded up to an integral number of pages for storage allocation purposes although the user specified quota limit will be enforced for object creation. The only effect of this is that some part of the last page of a storage area may never be used - a small price for the benefits of page based physical memory management.

4.5 The Implementation of AESOP's Basic Types

The discussion of this implementation of AESOP will be completed by describing the implementation of AESOP's basic types. The problems here tend to be simpler than logical/physical memory management since there are few difficult algorithmic problems.

Booleans, characters and integers are all represented in the references to the objects themselves so no storage is associated with them. The operations on these objects are trivially implemented. The only minor complications are that these types, and all of the built-in types, must check for correct access to perform the operation and to see that their operands are of the correct type.

Vectors must occupy storage so that their state may be remembered. The create operation creates the representation of the vector by employing the object creation algorithm described in chapter three. (In fact all of the remaining types will also do this when they need to create a representation for something.) The new_status operation, when it requires the lengthening of the vector, can either expand the vector in place, if the pattern of free storage permits, or move the vector and leave a forwarding pointer behind. The storage for vectors declared to contain only booleans or to contain only characters should be minimized since the use of a full object reference for these is not space efficient. This is accomplished by noting, in the vector's header, that it is one of these types of vectors and then allocating the minimum number of bits per element. This works since the uniform types of these vectors allows all type information to be kept in the object header.

Architecturally, a type manager is nothing more than an ordered set of procedure references. The implementation of type managers need be little more than that. Similarly, closures, procedures and sequencers are architecturally defined to consist of a few object references and their implementation needs do little more than store that information. In all of these cases, the operations are trivial.

The code segment type manager is the basic instruction interpreter of AESOP. This type manager is responsible for "compiling" the representation passed to code_segment$create into some lower level form most suitable to the AESOP implementation. Given this compilation aspect of the code segment type manager, it is clear that this type manager may become arbitrarily complex as various optimizations on the produced code are attempted. These result in increasingly complex implementations of AESOP and so are not all desirable given the economic constraints on a personal computer. However, all implementations of AESOP should compile instructions that specify a literal built-in type manager to be invoked along with a literal operation number and literal locations of the operands (or literal data) into very efficient low level code. This is possible since everything is known except the actual values of the operands. For example, the instruction:

```
integer$add(<lns, 7>, "9") return(<lns, 8>)
```

could be turned into a load from location 7 offset from a pointer register to the current LNS followed by an add of 9 and then a store to a location offset 8 from the LNS.[1] As many instructions will have this form, the result should be a reasonably efficient implementation of code segments.

The remainder of this section discusses the implementation of the io, storage area and process/event count type managers. In each case, the issues involved are more complex than for the preceding types.

4.5.1 The Implementation of the IO Type Manager

Chapter two presented the mechanism by which a program running on AESOP controls I/O devices. This was done by modelling each physical device as a logical device of type io that consisted of a set of mutable

[1] Of course, this example has ignored type checking. This makes it more complicated but the basic optimization remains.

device registers. This section discusses one way to implement this model.

An I/O device interfaces to the AESOP system by plugging into one of a number of plugs that are part of an AESOP processor. Internally AESOP numbers these plugs and the address of an I/O device, as specified in an io$create operation, is the number of the plug that the device is plugged into.

This section will specify the logical interface between the processor and the I/O device in the form of the commands that each can issue and the possible responses. This interface must be realized as an electrical interface at some point. However, the derivation of an electrical interface once the logical interface is specified is easy and is left as an exercise to the reader.

An io object consists of a sequence of event count, buffer and status registers. The processor maintains the first two in a block of storage that represents the device internally to AESOP since these registers refer to AESOP objects. The device maintains status registers since they reflect the status of the physical device.

The processor must be able to read and write the status register in the I/O device to implement the set_register and set_status_abcd operations. This is the only activity that the processor initiates. The set_status_abcd operation requires atomicity at the processor end (two users can not simultaneously perform such an operation on the same status register). This is accomplished by appropriate locking within the AESOP implementation - the device is not involved in the atomicity property (it may modify the register while the user is performing his operation).

The remaining activity on the interface plug is initiated by the I/O device either spontaneously or as a result of AESOP programs setting one of that device's status registers. This activity falls into two categories: accessing user supplied buffers (as specified by setting a buffer register in an io$set_register operation) and incrementing user supplied event counts to signal the occurrence of some event to the user.

All data transfers between the I/O device and a user supplied buffer are bracketed by the pair of device issued commands begin_dma and end_dma. The begin_dma command selects one of the bit vector buffers that the user has supplied to the I/O device and causes the processor to make that buffer accessible to the device, presumably by transferring the bit vector into primary memory and arranging that the device have access to that part of memory. The device can then read and write bits within the buffer, so long as the appropriate access to the accessed buffer is permitted. An end_dma command informs the processor that the I/O device has completed the transfer and the buffer may resume migration between primary and secondary memory. Only after the processor acknowledges the end_dma command may the device indicate to the user that the I/O has completed. This permits the data to be transferred between AESOP system buffers, if any, and user bit vectors.

To handle the problem of an I/O device that never completes an I/O sequence, and thus ties up a primary memory buffer forever, the AESOP implementation will associate a timeout with every buffer that has been selected by a begin_dma command. If there are no transfers to or from this buffer during the timeout period, the processor will unilaterally act as if an end_dma command had been issued. The next time the device attempts to perform a transfer, a suitable error will be returned to the device and the transfer not performed.

AESOP must handle the actual buffers used by the I/O device very carefully due to storage management problems. These buffers can either be a system supplied buffer or the actual bit vector supplied by the user. If a system buffer is used, it is necessary to copy the information from the user's bit vector to the buffer before the I/O begins (i.e. when the begin_dma command is issued) and from the buffer to the user's bit vector when the I/O completes (i.e. after the end_dma command is issued). This is adequate since the semantics of I/O are defined so that the contents of the user's bit vector are not guaranteed until after the end_dma command is issued and acknowledged by the processor. This is, however, inefficient since it requires copying all of the data between the user's buffer and the system's buffer.

Alternatively, the I/O device may deal directly with the user's bit vector. This requires that the system wire the user's bit vector into primary memory so that the I/O device can efficiently access the buffer. This approach, however, interacts in an unfortunate way with the storage management policies of this particular implementation of

AESOP. Consider what would happen if the bit vector resided in a storage area that was being garbage collected while I/O was being performed. Since the garbage collection algorithm being used in this implementation is a copying one, it is necessary, in general, to change the secondary storage address of the bit vector as part of garbage collection. In general such movement will mean that the position of the start of this vector within a primary memory page will change. This means that it is impossible to just rename the primary memory page with the name of a new virtual page, instead it is necessary to actually copy the bit vector to a new page. Such copying is not, however, compatible with efficient I/O since I/O may need to be stopped during such a copy.

Since neither of these buffer management strategies is adequate in and of itself, a composite scheme is used. When I/O begins, the user's bit vector is locked into primary memory and I/O begun using that bit vector directly. If it becomes necessary to copy that bit vector due to some storage management decision, that bit vector is copied to its new location but the old copy is left behind and used as a system buffer from this point on. When the I/O completes the contents of the system buffer are copied into the new location of the bit vector. This approach allows the most common case, a bit vector that does not have to be copied, to be handled very efficiently while allowing the handling of a moving bit vector in a graceful manner.

4.5.2 Storage Areas

Storage areas are the architecturally defined means for users of AESOP to deal with physical storage problems. The implementation of storage areas must first deal with the quotas that are the essence of storage areas, the count of used and free storage. A storage area must also manage its storage: object creation/deletion and garbage collection. These are just the logical memory management functions discussed in chapter three. Finally, storage areas must deal with the "close" relation and the inter-area cables that it implies. The difficulty is that cycles of cabled areas must not appear; otherwise the garbage collection algorithms of chapter three will fail.

It would be ideal to create cables whenever they do not create a cycle. Unfortunately, this determination is difficult to make both in space and in time since it amounts to computing the transitive closure of the cabled-to relation. New cables can be checked for legality, added and the new transitive closure computed in time O(N) using space $O(N^2)$ to hold the matrix that represents the transitive closure itself where there are N storage areas in AESOP. The time is not excessive. However, the space may be excessive (recall that there are expected to be several hundred storage areas in AESOP). However, if the number of cables is small, sparse matrix techniques can reduce the storage requirements so this may not be a serious problem. So far this looks feasible. Unfortunately, if a cable is deleted (by a storage_area$not_close operation), the transitive closure matrix must be recomputed which takes $O(N^3 \log N)$ time. If N=500 and performing one operation takes one microsecond, it takes over 10 minutes to recompute! Obviously this is unacceptable.

Instead, not all possible cables will be created - only those that are easily checked for validity will be permitted. A cable created to an area that has no outgoing cables can clearly never cause a cycle. So, this implementation uses the simple to check rule that cables may only be created to areas with no outgoing cables. Although this rule seems restrictive, it is not so when usage follows the subsystem model. In that model, cables are created only to library and subsystem areas. However, those areas are not cabled to anything so that the restriction of this rule is not a detriment to efficient operation.

If a cable is requested but can not be created under this rule, the storage area will note the request and, if later it can be granted, it will do so. This may permit the eventual creation of the requested cable as other cables are removed.

The storage_area$not_close operation requests, in this implementation, the removal of a cable. The cable can not just be removed, however, as there may be outstanding inter-area references that are valid because the cable is present. Instead, the cable is marked "deletion-requested". The next time the area cabled from is garbage collected, the garbage collector ensures that all references to the cabled area are by links, creating links if needed. Once the garbage collector is done, the cable may be removed as no references require it.

4.5.3 Processes and Event Counts

AESOP provides the user with the ability to easily create and use processes. To be useful, these processes must be cheap to use. This section discusses how to achieve this. Event counts are included here since they can cause processes to begin and cease execution.

Process efficiency comes in three forms: the expense of creating a process, the cost to choose a new process to run (scheduling costs) and the cost to actually switch a physical processor from one process to another. Process creation in AESOP is trivial. First a block of storage is allocated for the process object and initialized with the arguments passed to the create operation. An LNS stack is then created in the default storage area for the process. Finally, the newly created process is marked as being stopped and about to execute the proc$call_with_gns instruction that is the first instruction in every process.

Irocess switching in AESOP is also easy. The physical processor must know six things about a running process - current LNS and GNS, current instruction (a code segment and an instruction offset), default storage area and the location of the LNS stack. Process switching only involves storing these six values for the current process in its representation and restoring them for a second process. An associated cost of process switching is the page faults a newly started process tends to take. These can be minimized, if desired, by pre-loading the pages of that process before starting it as Multics originally did[37].

Process scheduling (i.e. choosing which process to run next) is almost as easy. The implementation is required to run the highest priority runnable process at all times so that a priority-ordered queue of runnable processes is the appropriate database for the implementation's process scheduler. Choosing a process to run involves searching this list, round robin within processes of the same priority, for a suitable process. When the priority of a process P is changed by a schedule operation, the queue must be reordered to reflect this. If the new priority of P is greater than that of the current process, P preempts the current process; otherwise no preemption occurs. When a process P becomes unrunnable for any reason, the queue is searched starting with P for a runnable process (since the highest priority process is always running, no higher priority processes need be examined). If a process P should become runnable (by a start operation or by a page fault P took being satisfied), it preempts the currently running process only if P's priority is higher than that of the current process. Finally, when an event count is incremented, one or more processes waiting on it become runnable. This set can be found by maintaining, with every event count, a list of processes waiting on that event count, in increasing order of awaited value. The highest priority process in this set of newly runnable processes is the only one eligible to preempt the currently running process. However, when a process becomes runnable in this way, it must be removed from the list of waiting processes on all of the event counts it is waiting on. Accessing those lists may cause page faults so that there may be a delay in giving the processor to a just awakened, high priority process.

Note, however, that AESOP need not be idle while satisfying these page faults - lower priority processes may be run.[1] Moreover, there is little performance impact if processes tend to wait on a single event count at once or if all of the event counts waited on are part of a single vector (in which case the garbage collector will tend to place them physically near each other, thus minimizing the number of page faults needed to access all of them).

Thus with proper implementation, processes in AESOP can be relatively cheap. They are easy to create. Process switching is trivial due to the small state of a process. Process scheduling is easy due to the round robin, priority scheduler defined by AESOP.

4.6 Hardware Considerations

The discussion of the last two chapters has presented a high level discussion of one possible implementation of AESOP. This discussion has been at a fairly high, hardware independent level. This section points out a few places where some special hardware assists could make the implementation of AESOP more efficient.

One of the most frequent activities of the implementation is determining whether or not a given page is in primary memory and, if so, where it is. A small associative memory containing the addresses of the

[1] The ability to run lower priority processes when page faults occur is why there is no scheduling cost associated with an ec$wait operation even though that operation may incur page faults to place the awaiting process on these same lists.

most recently referenced pages will be a significant aid to performance. In fact, a 16 entry associative memory on Multics that performs a similar function yields a 98% success rate as reported by Schroeder[45]. Moreover, since page identifiers are system-wide, this associative memory does not impose any overhead when the implementation switches between processes (i.e. there is no need to clear or save/reload its contents).

The implementation must map a secondary storage address to a storage area to determine the actions necessary upon copying a reference from one place to another. Due to the frequency of this mapping, there should be an associative memory of the most recent such mappings.

The implementation frequently needs information about storage areas. For instance, whenever an LNS is created/destroyed in procedure calls, the quota information must be updated. For this reason there should be a cache of the most recently referenced storage areas holding the needed information about those areas. Such information should include, at the least, the area's free storage information and its quota information. The utility of this cache will depend on how frequently programs actually tend to request object creation on AESOP.

The implementation, in general, is required to perform type checking at run time. As references to built-in types are expected to be most frequent, some hardware support to permit type checking of operands to the built-in type managers at the same time as those type managers are executing is important to efficient operation. The type manager should assume that its operands are of the correct type and begin the operation. Meanwhile the type checking is performed by a separate piece of hardware. If the type checking fails, the operation is aborted, otherwise the type checking has cost nothing. The operation, meanwhile, should make no modifications to objects until the type checking has succeeded.

The garbage collection of storage areas is a very important activity of the implementation. As described in this thesis, it has been done by the central processor in the manner suggested by Baker (*i.e.* incrementally). With the advent of cheap microprocessors, it is reasonable to place such activities in the memory itself and allow garbage collection to proceed in parallel with regular computations. This would require some modifications to the basic algorithms presented in the last two chapters, but should be possible with little problem. This should result in a more efficient implementation at some additional cost in complexity.

These are a few examples of the places in which specialized hardware can help the AESOP implementation. These represent speedups and are not essential. They should not significantly affect the cost of the implementation and so are reasonable to consider in an implementation.

4.7 Conclusions

This chapter completes the description of one implementation of AESOP. The major thrust of this chapter has been to bring the aspects of the physical hardware that AESOP will be running on to light. This chapter has discussed the management of the multi-level storage system that AESOP will be running on. The implementation chosen uses secondary storage addresses as the names of objects. Secondary storage is divided into a number of fixed sized pages and an object is brought into primary memory by bringing the page(s) that it resides on into primary memory. As primary memory becomes filled, classical page replacement strategies are employed to throw pages out of primary memory.

The problem of determining which storage area an address refers to is important since this question must frequently be answered due to the link/cable mechanism presented in chapter three. This question is answered by maintaining a compressed table that maps ranges of secondary storage addresses into the name of the corresponding storage area. To make this mapping simple the allocated size of storage areas is always rounded to an integral number of pages.

The major problem in allocating storage to storage areas is external fragmentation of secondary storage. Although storage may be wasted due to fragmentation, Knuth has found that the storage allocator is unlikely to ever be unable to allocate storage. This policy is not perfect as it is still possible for the system to be unable to satisfy an allocation due to fragmentation. If this unlikely event should occur, AESOP will come to a temporary halt while secondary storage is compacted. This is assumed to be such a rare event that this interruption of service is preferable to the complexity needed to handle it dynamically.

The implementation of the basic types has also been discussed. For the most part these were seen to be trivial. The I/O type manager is of importance since it specifies a physical interface to AESOP. Storage areas require care due to the need to prevent cycles of cabled areas. Finally, AESOP processes were shown to be inexpensive to implement.

This chapter concluded by listing some of the ways in which special hardware can aid an implementation of AESOP. Of particular importance are associative memories to remember the most recently performed table lookups.

Chapter Five

Using AESOP

Chapter two described the high level architecture AESOP. Chapters three and four demonstrated that AESOP can be implemented in an efficient manner. This chapter shows various ways in which AESOP can be used. First, some language features are examined and an implementation on AESOP shown. Second, a line printer example is presented that shows how to use some of AESOP's operating system features. The various examples presented in this chapter serve as paradigms for using AESOP.

In many ways AESOP is a very unusual architecture while in others it is rather ordinary. AESOP's basic data types (booleans, characters, integers and vectors) are not unusual, and AESOP uses traditional go-to's as its basic intra-procedural control flow mechanism. In both of these cases the language problems they solve are neither easier nor more difficult than on conventional architectures; the problems are basically the same in both environments. Thus such problems are ignored here.

However, AESOP has a number of unusual features; their use will be demonstrated in this chapter. AESOP provides an execution environment consisting of immutable code and two name spaces as a naming context, all provided as objects. AESOP directly supports the termination model of exception handling. Object viewers provide type extension, access restriction and access revocation in a uniform way. AESOP also contains a number of features normally associated with operating systems: processes, inter-process synchronization, storage management and I/O.

The first four sections of this chapter discuss some features of high level languages and how AESOP supports them. The aim is to show how AESOP's features can, in practice, be used to solve language implementation problems. Section one discusses extended types and the parameterized definition of procedures and type managers to demonstrate AESOP's type manager mechanism. Section two presents four examples of using AESOP's flexibility in naming and execution. Dynamic and static scoping are seen to be implementable in ways analogous to implementations on conventional architectures due to this flexibility. Algol call-by-name and CLU iterators show how AESOP's closures can be used to solve language problems related to the execution environment of code. A dynamic linking example shows how to use object viewers as an indirection mechanism to extend the flexibility of AESOP's naming mechanism. Section three shows how the continuation and termination models of exception handling, as typified by Mesa and CLU, can be implemented on AESOP. Section four examines some features of classical Algol-like languages that AESOP handles deficiently.

This chapter concludes, in section five, with a line printer example that demonstrates how to use some of AESOP's operating system features: processes, I/O and inter-process synchronization.

5.1 Extended Types and Parameterized Definitions

AESOP makes it easy to construct new procedures and types. This section gives an example of creating procedures and type managers on AESOP. Many languages provide type and procedure generators (i.e. procedures that return types and procedures as results). This section shows how to build these on AESOP. They appear here since the hardest problem of parameterized definitions is parameterization by type. CLU will be used for the examples although the results apply to other languages that support type extension and type/procedure generators.

5.1.1 Extended Types

Many recently designed languages permit the programmer to create new types, types beyond the set built into the language. These languages are frequently designed so that the language is type safe; all type checking can be performed at compile time. This section shows how the type extension and run-time type checking mechanisms of AESOP can be used to implement a language's type extension facility.

Figure 5.1 shows a CLU cluster that implements the type "foo"; it provides two operations, a and b, and has an internal procedure c. Creating the AESOP type manager for foo involves creating the code segments for a, b and c, then creating the template LNS for each procedure and finally creating the procedures a, b and c. The procedures a and b are then passed to tm$create to create the type manager foo shown in Figure 5.2. Note that procedure c is really an internal procedure to foo as only a and b may refer to it.

```
foo = cluster is a, b

       rep = ...

       a = proc(x:foo)
           ...
           y:rep := down(x)
           c(y)
           ...
           end a

       b = proc() returns(foo)
           ...
           y:rep := rep$create()
           c(y)
           x:foo := up(y)
           return(x)
           end b

       c = proc(w:rep)
           ...
           end c

       end foo
```

Figure 5.1. Foo, an example of a CLU cluster.

Foo may convert its objects between abstract type (i.e. type foo) and representation type (i.e. the type of object used by foo to represent foo objects) by using the *up* and *down* operations. Invoking *up* on a representation object produces a foo object; invoking *down* on a foo object produces a representation object. The implementation, on AESOP, of *up* and *down* within foo depends upon the intended environment of foo and the objects it provides. If foo and the objects it provides never leave the compile time type safe CLU world, then *up* and *down* simply represent different views of the representation object of foos - there is no need to protect the representation object from unauthorized

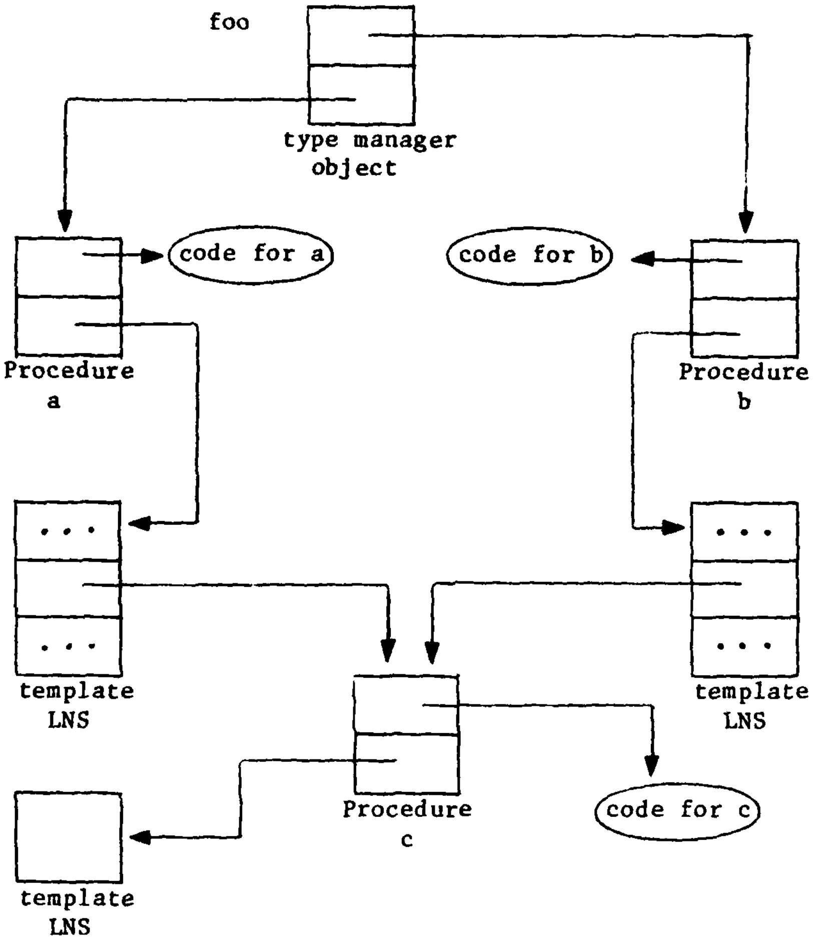
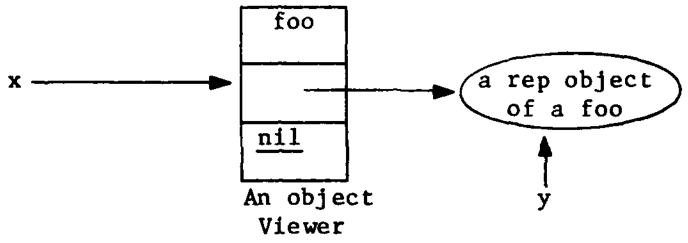


Figure 5.2. The AESOP type manager corresponding to cluster foo.

manipulation as the compiler can do this at compile time. Thus no code need be generated to perform up and down.

If, on the other hand, foo and foo objects are accessible beyond the CLU world, then foo objects must be protected from languages that are not type safe. The danger is that a foo object might be manipulated in ways that violate its semantics. Foo objects can be protected from such manipulation by making them into AESOP extended type objects by sealing the representation of a foo object in an object viewer. A sealed object and the code for up and down in this case are shown in Figure 5.3. Because only foo can perform up and down (due to the restrictions on using ov$seal and ov$extract), foo and its objects are protected from the actions of unsafe languages; only foo can perform down on an object that it has performed up on.

This protection is gained at some cost. First, extra space is needed for the object viewer sealing each foo object. Since object viewers are small (only three references), this is not a problem except for very small objects. Outside of foo, the use of sealing impcses no time penalty since only a reference is being passed around in any case.



```
 AESOP code for                          AESOP code for
y:rep := down(x:foo)                    x:foo := up(y:rep)


y := ov$extract(x) except(lose)         ov$seal(y,nil)
   ...
lose:
   process$signal("failure", ...)
```

Figure 5.3. The code for CLU's up and down operations.

Within foo, the expense is small. Only one instruction is needed to perform up or down and, typically, it only occurs at entrance to or exit from foo (i.e. only one up or down is performed per object per call). For all but the most trivial of operations these costs will be small in relation to the total cost of the operation. Thus the cost of this protection, in both space and time, is acceptable.

Only the person using AESOP can decide whether or not to seal foo objects. The ease of copying object references within AESOP means that it is extremely difficult, if not impossible, for either the CLU compiler or for foo itself to make this decision (except to always make the conservative choice of sealing). Thus the human user of AESOP must specify which option is to be used when creating a new type manager.

5.1.2 Parameterized Definitions

Many languages support parameterized generators of procedures and types (i.e. type managers). Conceptually, when supplied with parameters, they produce a procedure (type manager) as a result. Thus a procedure (type) generator defines a class of procedures (types) that are distinguished by the parameters given to the generator. This section will discuss only parameterized procedures since parameterized type are just a collection of parameterized procedures.

The most interesting and most difficult of procedure generators are those that are parameterized by types. For example, the procedure generator sort(T) might produce procedures that sort vectors of type T. Thus sort(real) would produce a procedure that sorted vectors of reals.

For concr^teness, the sort procedure generator will be used in the following discussion.[1] A name of the form sort_T will be used in the following discussion to name the procedure produced by sort when supplied with T as a parameter.

To sort its vector argument, sort_T must be able to compare the elements of its vector argument. Thus sort must require that a type T passed to it provide a compare operation on objects of type T so that sort_T can work correctly. The problem addressed in this section is the manner in which members of the family of procedures sort_T refer to the compare operations of the various T's. Although each T passed to sort will provide a compare operation, the operation index of compare may differ among the T's. This is the crux of the problem.

The procedures generated by sort can be implemented on AESOP in three ways as described by Atkinson[2] in relation to CLU. First, an entirely new procedure can be constructed for each invocation of sort. This results in a procedure that is maximally efficient; all dependencies on sort's parameter can be accounted for in the code for sort_T. Thus if the integer and real type managers provide compare operations with different operation numbers, the procedures sort_integer and sort_real will be specialized as needed.

[1] For expository simplicity, only single parameter generators such as sort are considered here. The extension to multiple parameter generators is simple.

The second possibility is to create sort_T as shown in Figure 5.4. In this case, the various procedures that sort might produce (sort_integer and sort_real are shown in this figure) share code which reduces space requirements compared with the previous scheme (each procedure was completely distinct there). However, there is a problem when sort_T invokes the compare operation on the objects it is sorting. In the code segment c, the instruction that invokes compare must specify the operation number of compare. Since the operation number of the compare operation can be different for the real and integer type managers, placing any constant operation number in c is incorrect.

This problem can be solved in two ways. First, it could be required that the type passed to sort use certain operation numbers for certain semantic operations (e.g. operation one must be the compare

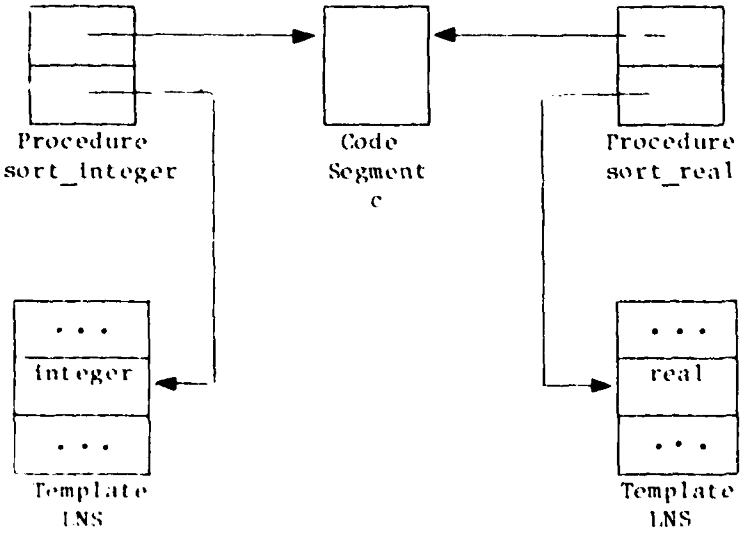


Figure 5.4. A parameterized procedure instance in AESOP.

operation). This has the disadvantage of restricting the types that can be passed to sort. However, this can be programmed around by creating a new type manager that maps the operation numbers required by sort into the operation numbers of some other type manager. This is undesirable as it involves the creation and use of an otherwise unnecessary type manager. The second possibility is to pass an extra argument to sort that specifies the mapping of operation numbers from those sort desires to those sort′s argument provides. In this case, the template LNS for each sort_T would retain a reference to (a copy of) this mapping for use at the time the procedure is running.

Finally, the procedure sort_T(<args>) could be mapped into the procedure sort′(T, <args>). In this case there is only one procedure, sort′, that implements all of the sort_T′s so that space usage is minimized (the previous two schemes had a procedure for each sort_T). On the other hand, an extra argument must be passed on every invocation of a sort_T which increases the space and time overhead associated with calling a sort_T. Moreover, the problem of invoking, within sort′, the correct semantic operation on T arises once again. Thus, either T must be restricted to a certain class of types or a second parameter must be passed to sort′ that maps operation numbers. Both choices further reduce the desirability of this scheme.

The characteristics of the family of sort_T′s determines which of these three schemes should be used. If there are only a few sort_T′s (i.e. sort itself is invoked only a few times) but they are called frequently, then the first scheme′s low execution overhead is desirable.

If there are many sort_T's that are called infrequently, the third scheme's low space overhead is desirable. In all other cases the second scheme represents a reasonable compromise.

5.2 Using AESOP's Flexibility in Naming and Execution

AESOP provides flexible means for creating and manipulating the components of execution (code segments and name spaces). This flexibility is useful in implementing those language features that require special handling of a program's environment. Moreover, AESOP's object viewer mechanism can be used to provide a general indirection facility in naming on AESOP.

This section presents four examples to demonstrate these points. First, the problems of dynamic and static scoping are addressed. Second, the use of closures to implement Algol-60's call-by-name parameter passing demonstrates how to dynamically create and modify naming environments and how procedure-like entities might be passed as arguments in AESOP. Third, the implementation of CLU iterators on AESOP shows how coroutine-like structures can be created using closures. This further demonstrates the use of closures for passing procedure-like entities as parameters. The fourth example shows how to implement dynamic linking by using object viewers as an indirection mechanism and using exception handling to detect breaks in the indirection chain.

5.2.1 Dynamic and Static Scoping

In AESOP, a variable is a slot in an LNS. A procedure may directly refer only to entries in its own LNS. Thus AESOP provides no non-local scoping for binding variable names. However, AESOP provides facilities that are powerful enough to support classical implementation techniques for both dynamic scoping (as in Lisp) and static scoping (as in Algol-60).

Dynamic scoping entails binding a free variable to the first variable of that name found by tracing backwards through the call chain. In general, this can only be done in an interpretive manner due to its dynamic nature. AESOP provides no help here beyond the ability to build and maintain any appropriate databases. The ability to treat LNS's as objects in AESOP is, however, essential to making these techniques work.

Static scoping entails binding a free variable, at compile time, to a variable in a statically enclosing scope. At run-time the correct instance of this scope must be found when referring to that variable. AESOP allows any of the classical implementation techniques, such as displays, to be used to do this since AESOP allows naming environments to be treated as objects and thus stored in databases and passed as parameters.

5.2.2 Algol-60 Call-by-name Parameters

Algol-60's call-by-name parameter passing mechanism is a more complicated parameter mechanism than the normal AESOP parameter mechanism as it permits arbitrary computation upon each reference to an argument. The implementation of call-by-name parameters on AESOP is a second example of using AESOP's flexibility in naming and execution, in particular the use of closures.

Call-by-name parameters can be implemented on AESOP in the same way as they are provided in many Algol implementations - by "thunks"[39]. Every call-by-name parameter to a procedure P is replaced with a pair of parameters, both closures, one of which performs references to the parameter and the other which performs assignments to the parameter. Closures are passed instead of procedures both as an efficiency measure to avoid the overhead of creating a new LNS for the thunk on each use and as it is the simplest way to permit the thunk to execute in the correct naming environment. Parameters are passed to the thunk through a vector "value" that is also passed to P. Thus the procedure P = proc(a:by-name)[1] is transformed by the Algol compiler into:

P = proc(value:vector, a_get:cl, a_set:cl)

P refers to "a" by executing a closure$run(a_get) operation. The closure a_get, whose encapsulated naming environment permits it to run

[1] Single parameter procedures are considered here for expository simplicity.

in the environment required by Algol semantics, computes the value of a and returns the result. In essence, a_get is a parameterless procedure that returns a's value as far as P is concerned.

To perform a:=x, P invokes a_get by passing, in value, x. That is, P executes:

```
value(1) := x
closure$run(a_set)
```

The closure a_set, whose encapsulated environment permits it to run in the correct environment, modifies the value of a in P's caller to be the same as value(1). Essentially the shared object "value" is being used to pass parameters from P to the "procedure" a_set.

Alternatively, P could be given the components of a_set as arguments (i.e. a_set_code, a_set_lns and a_set_gns). P can now pass parameters to a_set by setting conventional locations in a_set_lns and a_set_gns. P then invokes a_set by first creating a closure out of these three components and then invoking closure$run on that closure. This scheme has the advantage that a_set may directly refer to its parameters in its LNS and GNS instead of copying them into its LNS and GNS from another vector (i.e. from the vector "value" in the above example).

5.2.3 Iterators

CLU provides iterators as a control abstraction. They allow a piece of code, the body of a for loop, to be executed with differing values of its loop control variables as yield'ed by an iterator. This section uses CLU iterators as the canonical example of iterator-like programs. Two factors concerning iterators are of particular concern: the body of a loop must have access to the variables in the procedure containing it and the iterator must retain its state after *yield*'ing a value.

Since iterators and *for* loops have coexisting environments, a coroutine implementation seems natural. AESOP, however, does not provide a coroutine mechanism. Instead, iterators and *for* loops must be implemented using the hierarchical control mechanisms of AESOP.[1] A key to the correct functioning of the implementation below is the CLU restrictions that result in nesting of iterators.

One possible implementation of iterators is the following. A CLU iterator, I, of the form:

```
I = iter(a(1):T(1), ..., a(N):T(N)) yields(T(R1), ..., T(Rm))
    ...
    end I
```

is transformed by the CLU compiler into the AESOP procedure:

```
I  =  proc(body:closure,   comm_vec:vector[1:m],   a(1):T(1),   ...,
   a(N):T(N))
   ...
   end I
```

where body is a closure consisting of the code that is the body of the *for* loop and its naming environment (i.e. the LNS and GNS of the

[1] Theoretically processes and message passing could be used. However, even though AESOP processes are cheap, they are not cheap enough to permit creation of a process for every *for* loop.

program containing the for loop) and comm_vec is used by I to yield values to the loop body.[1] Inside I, the CLU yield statement:

```
yield(o(1), ..., o(m))
```

is transformed by the compiler into the AESOP code:

```
comm_vec(1) := o(1)
...
comm_vec(m) := o(m)
closure$run(body)
```

Thus the iterator yields values to the body of the loop through the communication vector comm_vec and the loop body runs in the correct environment (so long as body was correctly constructed). Executing the for loop simply involves creating the correct closure "body" and invoking I with the appropriate arguments.

The body of a for loop may do one of four things while it is executing: return to the iterator for new objects (by falling off the end of the loop or by executing a continue statement), break from the loop (causing execution to continue at the statement following the for statement) or cause the procedure containing the loop to return (either normally or abnormally). In AESOP, a return to the iterator occurs when the body executes a process$return operation. This causes the iterator to continue immediately after the closure$run operation that caused an iteration of the loop body to be run. The iterator may then yield new objects to the body of the for loop or it may return to its caller (thus terminating the CLU for statement). The other three activities of the

[1] The loop body accesses comm_vec using a reference that the procedure invoking I placed in the LNS in body.

loop body, however, require the cooperation of the iterator since they embody the concept of a non-local go-to, something AESOP has deliberately prevented since it is a source of many program errors. Non-local go-to's are implemented by having the body abnormally return to the iterator with the break, return or signal exception to indicate the corresponding request of the loop body. For a break exception, the iterator returns to its caller, thus terminating the loop. For the return and signal exceptions, the iterator resignals the exception to its caller who performs the appropriate action (i.e. a normal or abnormal return).

Thus control structures similar to CLU iterators are easily implemented on AESOP. This example shows the utility of using closures to package a piece of code with the naming environment in which it should eventually run. This implementation of iterators is fairly efficient since yielding to the body, the frequent inter-procedural transfer of control in this implementation, only involves pushing a small activation record onto the control stack (recall the implementation in chapter three). The storage requirements are small: the one time creation of the loop body's code segment and the dynamic need to create closures. Moreover, the implementation presented here has two additional advantages.

First, this implementation allows the iterator to clean up when the iteration is about to terminate since only it can terminate the for loop (i.e. by returning); the loop body can only request termination of the iteration. This is useful in an environment with parallelism such as in AESOP where a data item might be shared by concurrent processes. If an iterator-like program has locked a database so as to yield a consistent set of objects from that database, then it must unlock that database before being terminated (otherwise deadlock will eventually result).[1]

Second, AESOP assumes that mistrust may be present. Suppose that an iterator-like program is not trusted so that its potential damage needs to be controlled; that is, it may only modify objects accessible through its arguments, the a(i) in the above example, and not through the extra information passed as part of the implementation of iterators (the body and comm_vec arguments in the above example). This is ensured by this implementation of iterators since the environment of the body, which might be a gateway to the world, is inaccessible to the iterator because closures are inviolate. If comm_vec is initialized to nil or passed to the iterator as write only (by using an object viewer for access restriction), then the iterator can never access anything through comm_vec. The iterator's potential to cause damage is thus severely limited.

5.2.4 Dynamic Linking

An unusual use of AESOP is to implement dynamic linking. This example further demonstrates AESOP's naming flexibility by showing how object viewers can be used as an indirection mechanism.

[1] Note that this issue never comes up in CLU; since there is no parallelism to deal with, no locking is needed.

Dynamic linking permits the binding of a character string procedure name, say S, to an actual procedure to be delayed until the first time the procedure containing the reference to S attempts to access the procedure "named" S. At that time, S is bound to an actual procedure, call it P, which is then called.

The scope of the binding of S to P is one of the means of classifying dynamic linking mechanisms. In Multics[37], this binding is for a given process, other processes are unaffected by this particular binding.[1] Other durations of this bindings are possible, such as binding S to P for an entire subsystem or only for this particular call on S in this invocation of Y. In this section, the term scope (or binding scope) will refer to the range of invocations over which a particular binding of name to procedure is to be valid. The implementation presented in this section will permit efficient binding for various scopes.

This section develops a dynamic linking facility for AESOP in two stages. First a very simple mechanism will be presented. The flaws in this scheme will be pointed out and corrected. The dynamic linking facility that results will be efficient and will allow links (i.e. calls with character string names that need to be bound to an actual procedure) to be snapped for a given scope.

[1] This is not strictly true since the user can take actions to prevent the binding from being per-process. However, this technicality is unimportant for the current discussion.

Consider a procedure Y that needs to contain a "call x(...)" statement where the name "x" is to be dynamically bound. Suppose that Y's code segment contains the instruction:

```
proc$call(<lns, i_x>, ...)
```

where i_x is an integer associated with calls on x by Y. The dynamic linking facility must ensure that the correct procedure for the current scope is called whenever this instruction is executed.

Consider the procedure Y in Figure 5.5. When the call instruction is executed, an unexpected_nil_operand(1) exception will occur since entry i_x of Y's LNS contains nil as its object reference and not the name of a procedure. If the code around the proc$call is as in Figure 5.6, the unexpected_nil_operand(1) exception that occurs will cause this code to call auxiliary procedure resolve_name to find a procedure, call it P, that is to be bound to the name "x" in the current scope. (Resolve_name will maintain a data base, if necessary, to ensure that this binding is maintained for the duration of the current scope). The

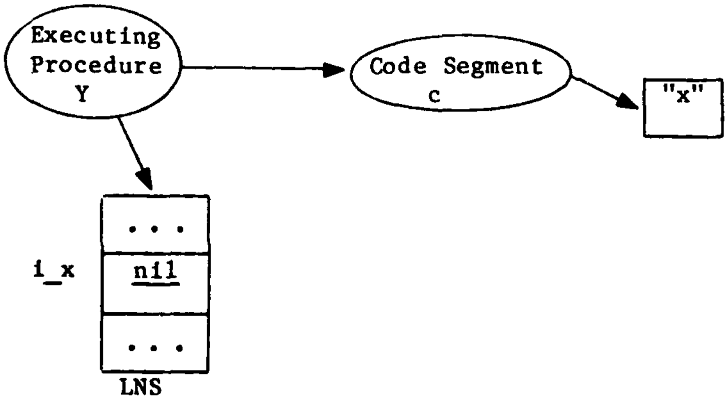


Figure 5.5. Executing Procedure Y before snapping the link.

```
instr:proc$call(<lns, i_x>, ...) normal ok

        % Let S be the signal name
        % Let Op be the signal operand

        % Is it a linkage fault? (Note that we are only interested in
        %   the first parameter to proc$call since it is the procedure
        %   to be invoked.)
        if S ^= unexpected_nil_operand then go to other
        if Op ^= 1 then go to other

        % It is a linkage fault so resolve it
        % Resolve "x" to an actual procedure by calling resolve_name.
        %  Use the returned procedure to snap the link.
        <lns, i_x> := resolve_name("x")

        go to instr
other:...
   ok:...
```

Figure 5.6. The code to handle a linkage fault.

i_x'th entry in Y's LNS is changed to a reference to P so that, when reexecuted, the proc$call at instr proceeds correctly.[1] After this exception handler executes, the situation will be as in Figure 5.7. Note that the link, which is the i_x'th entry in Y's LNS, has only been snapped in this invocation of Y. All future calls on "x" by instructions using entry i_x of this LNS will proceed very efficiently with no interpretation involved. All calls on "x" in other invocations of Y (current or future) are unaffected.

[1] In a more general situation where the link is being snapped to an arbitrary object and passed to an arbitrary invocation, reexecuting the instruction always works for the built-in types (they check their arguments before doing anything). If user written programs clean up before returning the unexpected_nil_operand exception it is also true for them.

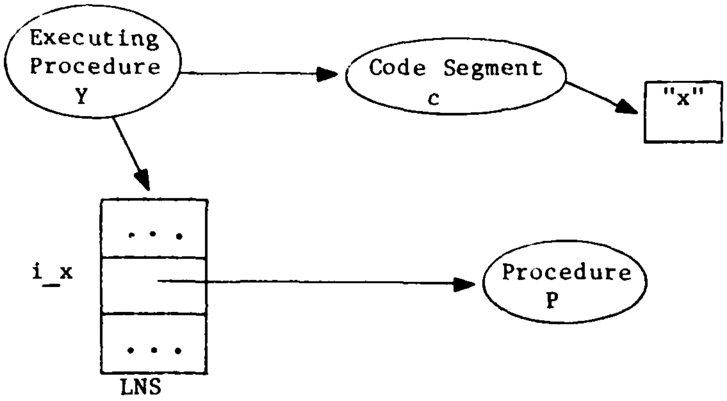


Figure 5.7. Executing Procedure Y after snapping the link.

Note that binding x to the correct procedure in the current scope is easily done with this mechanism since the link is resnapped in every invocation. The problem with this scheme is efficiency since a link must be snapped in every new invocation. This amounts to an interpretive solution of unacceptably high execution time overhead. An alternative is needed.

An efficient dynamic linking mechanism will do two things. First, a link will be snapped to the correct procedure for the executing scope (i.e. a correctness requirement). Second, the number of times that a particular name must be bound to the procedure it represents will be minimized. This will minimize the number of times links to the procedure a name represents must be snapped. The second is accomplished by using object viewers as an indirection mechanism while the first is ensured by limiting those object viewers to a single scope.

AD-A083 433 MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTE--ETC F/G 9/2
THE ARCHITECTURE OF AN OBJECT BASED PERSONAL COMPUTER.(U)
MAR 80 A W LUNIEWSKI N00014-75-C-0661
UNCLASSIFIED MIT/LCS/TR-232 NL

3 OF 3
AD
A083433

END
DATE
FILMED
6-80
DTIC

These goals are met by causing Y to execute as shown in Figure 5.8 where O is the link. If the ? in O is nil then the link is unsnapped since a proc$call(<lns, i_x>, ...) instruction produces an unexpected_nil_operand(1) exception. In this case, the handler for the linkage fault will use O′ to set the object field in O to the procedure returned by resolve_name, thus snapping the link. The code in the handler is then:

```
p_var := resolve_name()
ov$modify(<lns, r_x>, p_var)
```

where r_x is an integer associated with calls on x by Y and p_var is a temporary variable. If the ? in O refers to a procedure, the proc$call instruction proceeds normally.

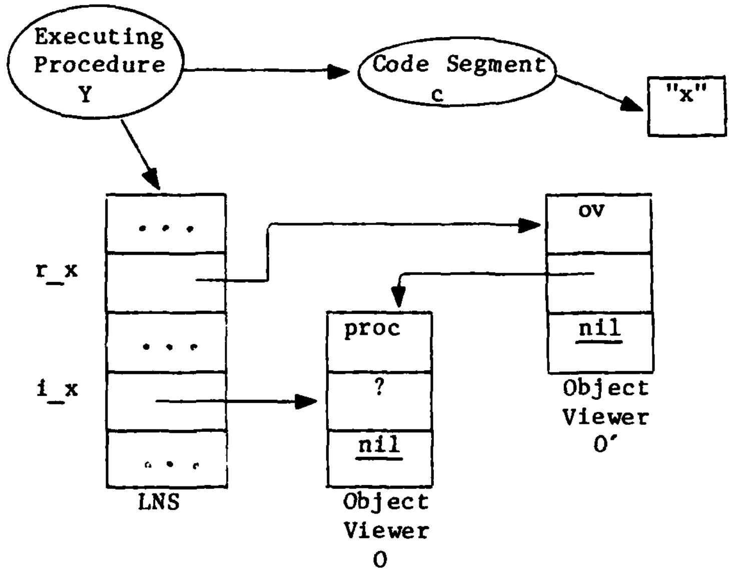


Figure 5.8. An executing procedure Y that binds names correctly.

To ensure that the link O is per-scope and thus snapped in only a single scope, two things must be done. First, Y must only be called in a single scope (by making copies of Y in all invoking scopes). Second, both O and O′ must be per-scope to ensure that the link O is per-scope. However, it is still desirable for all of the copies of Y to share the code segment c. This does effect per-scope binding if the shared data, c, contains no direct references to the link or link snapping information.

To make Y per-scope, a special object, of a new type tproc, is stored in the file system. Y will be represented as a tproc as shown in Figure 5.9. Resolve_name must now perform some additional work when it decides to bind a tproc from the file system to a name within the current scope. When resolve_name decides to use tproc Y in some scope,

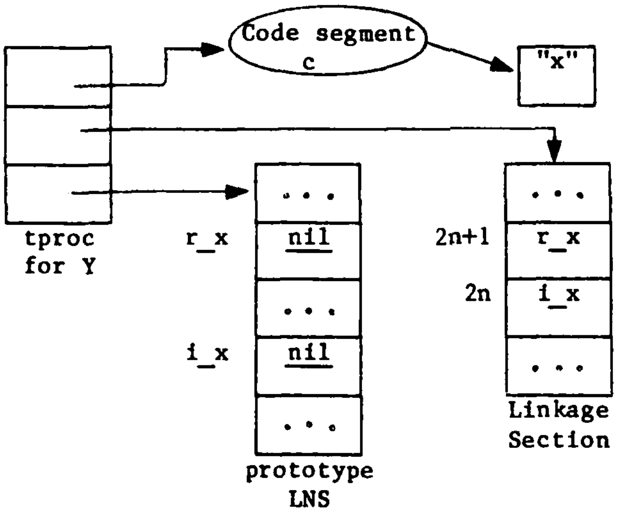


Figure 5.9. The tproc for Y.

it uses the tproc for Y from the file system to produce the procedure Y of Figure 5.8 as follows. First, resolve_name copies the prototype LNS into a new vector v. Then, for each pair of entries in tproc Y's linkage section, it creates an object viewer, call it O, with <u>nil</u> as its object field and a second object viewer, call it O', that permits modification of O. In the case at hand, resolve_name then does:

```
% Let ls be the linkage section for the tproc for Y
v(ls(2n)) := O
v(ls(2n+1)) := O'
```

This permits Y, when invoked, to refer to a procedure indirectly through an object viewer (i.e. through a link) and to modify that object viewer for the purpose of snapping the link. Once object viewers have been created for all pairs of entries in the linkage section, resolve_name creates the procedure Y of Figure 5.8 using c as Y's code segment and v as its template LNS. By creating O and O' at this time, it can be ensured that all invocations of Y in the current scope will have a reference to O and thus have x bound to an actual procedure the first time that the name x is used by an invocation of Y. Thus the link O will only be snapped once per scope. Moreover, if resolve_name allocates the object viewers for the link to the procedure named x only once and makes the references in the template LNS's of procedures such as Y refer to these shared object viewers, then every procedure within the scope using the name x will have the name x bound the first time that <u>any</u> procedure within the scope references x.

With this last modification, the dynamic linking mechanism is complete. First, and foremost, it is per-scope - a link snapped in one scope does not effect, in any way, the bindings in other scopes. Second, it is efficient. Once a link associated with the procedure Y is snapped within a scope, it is snapped for all current and future invocations of Y so the relatively expensive link snapping process is not repeated unnecessarily. Moreover, if multiple procedures share a link, snapping this link once snaps it for all of those procedures thus further improving efficiency. Third, dynamically bound references impose only the small overhead of an indirection through an object viewer. Fourth, the name resolution mechanism can be very flexible since the linkage fault is handled at the site of the fault. By passing additional information to resolve_name (or, in fact, by using different name resolution procedures at every proc$call site), it is possible to make the name resolution mechanism dependent upon the dynamic site of the fault within a scope (e.g. it can depend upon which process is running, which procedure is running and upon the actual call within the procedure that produced the fault). Fifth, the link can be unsnapped by using O′ to modify the object field in O to nil. This is a feature occasionally found useful when a name has been bound to the incorrect procedure or needs to be rebound for any reason.[1]

[1] Multics, for instance, uses this facility to unsnap links to procedures that are recompiled.

5.3 Exception Handling on AESOP

An important attribute of many recent languages is the inclusion of an exception handling mechanism. Such languages cause users to be aware of unusual conditions simply because an exception handling mechanism is there. As a result, users write programs that are more robust in the face of failure than they would have been otherwise. There are two models of exception handling: the termination model and the continuation model. This section will consider each in turn and show how they might be implemented on AESOP.

5.3.1 The Termination Model

In the termination model, as typified by CLU, a procedure may return either normally or abnormally. The caller continues at the statement following the invocation if the return was a normal one. Otherwise, the caller receives control in some exception handler that is responsible for dealing with the exception.

The set of handlers in a CLU program P is known at compile time; given an invocation within P and a possible exception returned by that invocation, the correct handler to invoke is known to the compiler. At run time it is only necessary for P to determine the exception and branch to the correct handler. Once in control, the handler may do some processing and then either perform a local transfer or cause P to return, either normally or abnormally.

AESOP contains an explicit mechanism for coping with the termination model - the ability to cause a forced go-to whenever an invocation returns abnormally. For every invocation within a program, the compiler causes abnormal returns to go to a handler that is responsible for handling all exceptions from that invocation. This handler transfers to the code that corresponds to the handler for the particular exception raised by the abnormally returning invocation. This is nothing more than a simple series of tests and transfers. The resulting implementation is cross between the branch table and handler table implementations proposed for CLU[2]. Within a handler the implementation of local transfers is trivial since AESOP provides go-to's. The handler simply does a process$return or process$signal operation to cause the procedure it is part of to return.

Thus the termination model of exception handling is easily handled by AESOP. Fundamentally, AESOP has provided a built-in means for terminating the signaller and entering a handler with the exception name and operands as implicit arguments whenever an exception occurs. The program must still discriminate based upon the identity of the exception.

5.3.2 The Continuation Model

In the continuation model, as typified by Mesa, a handler for a signalled exception is found by tracing backwards through the call chain until an appropriate handler is found. That handler is then invoked. The handler, perhaps after some processing, may then either resume or terminate the signalling procedure. If the signalling procedure is terminated, control continues in the procedure that contains the handler.

The first problem in implementing continuation model exceptions is discovering the handler to invoke. There are two ways to do this. First, a per-process database of enabled exception handlers, a condition stack, could be maintained. Whenever a handler is enabled, an (exception name, handler) pair is pushed onto the condition stack. When a handler is disabled, the corresponding entry on the condition stack is popped.[1] Signalling an exception involves searching the condition stack in LIFO order until a handler is found for the exception being raised and then invoking that handler. The execution cost of this method is small since only those procedures that enable handlers need be concerned with the whole mechanism. However, this method has the disadvantage that a per-process database is needed.

Alternatively, a special handler can be passed as an additional parameter with every procedure call. Whenever a procedure needs to raise an exception, it invokes the handler, H, passed to it, giving it the exception name and operands as arguments. H may then handle the exception or it may continue signalling the exception by invoking the handler passed to the procedure that H is associated with. In this way, the exception propagates backwards through the call chain until an appropriate handler is found. This method has the advantage of no

[1] Mesa guarantees this LIFO ordering.

per-process database but the disadvantage that every procedure must pass and accept this extra parameter. This is basically the implementation for Mesa outlined in the Mesa manual[36].

The first scheme is preferred when there are relatively few handlers as most procedures can then ignore the entire issue of exception handling. The second method is preferred when there are many handlers which are frequently enabled/disabled since there is no need to constantly manipulate the condition stack.

There are two other issues of concern in implementing the condition model: the form of a handler and the means by which handlers terminate the signalling procedure. Handlers must be able to access the environment of the procedure they are associated with as well as the name and operands of the exception currently being signalled. The handler is thus basically a procedure that executes in a special naming environment. It is implemented in either of the ways that the procedure that performed Algol call-by-name assignments in section 5.2.2 was performed.

The signaller is terminated by making a non-local go-to out of the exception handler to the procedure that contains the handler. They are implemented analogously to go-to's out of the body of a <u>for</u> loop, that is, the handler returns abnormally (using AESOP's built-in exceptions) with an unwind(target) exception. All procedures will resignal this exception until the procedure being transferred to (i.e. target) gains control. At this point, that procedure performs the actual transfer, now a local go-to. Note that a handler for the unwind condition may perform a local clean-up before passing the exception along. Moreover, by refusing to propagate the unwind exception, a procedure terminates the non-local transfer as allowed by Mesa.

Thus the continuation model is seen to be easily implemented on AESOP because of AESOP's naming and execution flexibility. This implementation is analogous to implementations on conventional architectures. Thus AESOP neither helps nor hinders the implementation of the continuation model.

5.4 Some Deficiencies in Handling Classical Languages

AESOP has been designed to support a certain class of languages very well. This section presents some ways in which AESOP is deficient in supporting classical languages such as Algol or Fortran. Two issues are discussed: the unnecessary cost of garbage collection and the difficulty of doing call-by-reference parameters.

5.4.1 Garbage Collection Costs

Many classical languages have either a stack or a static storage semantics. They have no need for a heap in their implementation so that AESOP's heap storage may reduce performance.

Scalar types (e.g. booleans and integers) are handled efficiently by AESOP since the variables (i.e. LNS slots) that refer to them contain the value of the object (recall the implementation of object references in chapter four); no storage is ever allocated for these objects. Aggregates (e.g. arrays and records) are another matter.

A record is nothing more than an array of values of differing type with the character string selectors being mapped by a compiler into integer indices. Thus only arrays will be considered here. An array is an AESOP object that resides in the heap. Thus, even though an array is inaccessible after the procedure that created it returns, the implementation of AESOP's heap proposed in this thesis means that it's storage will not be reclaimed then - it will be reclaimed later by the garbage collector.

Thus the AESOP heap results in inefficient use of storage (inaccessible arrays) and in wasted time (running the garbage collector to retrieve objects "known" to be inaccessible). Storage inefficiency can be reduced by having each procedure explicitly destroy its arrays just before returning. This reclaims most of the storage for arrays at the earliest possible time (only the tombstone left behind after object deletion by the AESOP implementation will not be immediately reclaimed). The cost of running the garbage collector is, however, unavoidable in the long run as tombstones accumulate.

5.4.2 Call-by-reference Parameters

Many languages use call-by-reference for parameter passing. This allows the called procedure to directly read and write an array, variable, array element or record component of its caller. This semantics is difficult to achieve efficiently on AESOP.

AESOP's basic parameter passing mechanism is what CLU has termed call-by-sharing; a formal parameter is caused to refer to the object that an actual argument refers to. Call-by-reference is achieved by passing an argument that permits accessing an array or a particular entry in an LNS or vector (i.e. a variable) so that the called procedure may access and manipulate the caller's variable. The difference between these two parameter passing mechanisms is illustrated in Figure 5.10.

Passing arrays by reference is trivial on AESOP since only the name of the array is being passed and this is AESOP's basic parameter passing mechanism. The passing of variables (or array/record components) is, however, non-trivial since AESOP provides no means of passing vector entries as arguments.

Variables can only be passed by reference interpretively in AESOP since variables are not objects in AESOP. Every procedure P = proc(a:by_ref) must be transformed into the procedure:

P = proc(a_vec:vector, a_offset:integer)

where entry a_offset of a_vec is the variable a of P's caller. P

```
P = proc                   Q = proc(i:int)
    a:int                      print i
    a := 7                     i := i+1
    Q(a)                       end Q
    print a
    end P

If i was passed by:          the output would be:
call-by-reference             7   8
call-by-sharing               7   7
```

Figure 5.10. An example of call-by-reference and call-by-sharing.

manipulates a by reading and writing entry a_offset of a_vec. This works semantically but is inefficient since any attempt to use "a" results in copying a's object reference into P's LNS. That is, if a is an integer variable, then the assignment a := a+7 turns into:

```
temp := a_vec(a_offset)
temp := temp+7
a_vec(a_offset) := temp
```

The resulting inefficiency of this interpretation in both space and time can be high. Moreover, all of a_vec is accessible to P so that P can, if malicious or undebugged, cause damage to P's caller through a_vec.

5.5 Using AESOP's Operating System Features

This chapter concludes with an example of how to use some of AESOP's operating system features: I/O, processes, inter-process communication/synchronization and protection. The example chosen is a line printer driver - a program that prints a vector of characters on a printer.

This section shows how a line printer driver might be implemented in AESOP. This is a simple driver, implemented as a new type lpt, that accepts a vector[char] that is to be printed and places the characters on the printer as soon as it is the turn of the current process to access the printer.

There are assumed to be two types of physical line printers, both to be supported by lpt. The first accepts characters one at a time for printing. The second accepts all of the characters of a vector at once for printing. Every line printer w!ll consist of a status re?ister (logical device register one) and two event count registers (logical device registers two and three). The second type of printer also has a buffer register (logical device register four). Bit one of a printer's status register tells which type of printer it is. If true, the printer accepts 8 bit characters one at a time in bits three through ten of register one. Otherwise, the printer prints all of the characters in the buffer named in register four by picking up the characters in that buffer with 8 bits per character. In either case, the setting of bit two of register one to true initiates the printing by the physical printe·. The event count in register two (three) is incremented by the printer whenever an operation completes successfully (unsuccessfully).

The line printer cluster, which provides the logical line printer device and which interfaces with the physical printer, is shown in Figures 5.11, 5.12, 5.13 and 5.14. Three high level comments are in order. First, the procedure char_to_bits (chars_to_bits) is not shown here as it simply converts a character (vector[char]) to a vector[boolean] representation. Second, these programs ignore the possibility of error returns from the built-in type managers for expository simplicity since the consideration of such errors would needlessly complicate the programs. Finally, every statement in these programs corresponds to one or two AESOP instructions even though some syntactic sugaring has been used (e.g. vector references and if

```
lpt = tm is create, destroy, print

     rep = vector[1:4] % first entry is an io object
                       % second entry is an event count
                       % third entry is a sequencer
                       % fourth entry is a revoker for the
                       %   object viewer sealing an lpt

     create = proc(addr:integer) returns(lpt)

              % Create the rep object
              v := vector$create(4, nil, nil)

              % Initialize v
              v(1) := io$create(addr, [1, 1, 2, 2, 3])
              v(2) := ec$create()
              v(3) := sequencer$create()

              % Seal v and return the sealed object
              a, v(4) := object_viewer$seal(v, nil)

              return(a)
              end create


     destroy = proc(l:lpt) signals no_access

                % Unseal l
                v, ar, a := object_viewer$extract(l)

                % Check for sufficient access to destroy l
                if ar(2) = true then go to no_access

                % destroy the actual io object
                io$destroy(v(1))

                % Destroy l and we are done
                ov$destroy_viewed(v(4))
                return

           no_access:
                signal no_access
                end destroy


     print = % see Figures 5.12, 5.13 and 5.14
        end print
     end lpt
```

Figure 5.11. The lpt cluster.

```
print = proc(l:lpt, c:vector[char])
   signals no_access, error

         % Unseal l
         v, ar := ov$extract(l)

         % Check for sufficient access to print
         if ar(3) = false then go to ok
         signal no_access

         % Wait until it is our turn to use the printer
      ok:ec$await(v(2), sequencer$take(v(3)))

         % Set the event count registers in the device
         el := ec$create()
         e2 := ec$create()
         io$set_register(v(1), 2, el)
         io$set_register(v(1), 3, e2)

         % See what type of LPT it is and call the appropriate
         %   procedure to actually print the vector c.
         bv := io$read_register(v(1), 1)
         if bv(1) = true
            then call unbuffered(v(1), c, el, e2) except error
            else call buffered(v(1), c, el, e2) except error

         % release l and return normally
         ec$increment(v(2))
         return

         % release l and return abnormally
   error:ec$increment(v(2))
         signal error
         end print
```

Figure 5.12. The print procedure of the lpt cluster.

statements).[1] Thus the simplicity of this program directly reflects the simplicity of the AESOP implementation of a line printer driver.

[1] Multiple AESOP instructions are needed only where temporary variables are needed to hold the results of an invocation for passing to a subsequent invocation.

```
buffered = proc(device:io, c:vector[char], e1:ec, e2:ec)
         signals error

         % Vector c is printed on printer device which is assumed
         %   to accept a whole buffer at once.
         % Event count e1 (e2) is the event count that the printer
         %   will signal normal (abnormal) completion on.

         % Convert c to bits and give to the printer
         io$set_register(device, 4, chars_to_bits(c))

         % Set bit 2 of register 1 to start the printer
         io$set_status_1111(device, 2, true, 2, 1)

         % Now wait for the printer to complete and
         %    check to see if an error occurred
         if 2 = ec$await(e1, 1, e2, 1) then signal error
         return
         end buffered
```

Figure 5.13. The auxiliary procedure buffered.

```
unbuffered = proc(device:io, c:vector[char]) signals error

       % Vector c is printed on printer device which is assumed
       %   to accept characters one at a time.
       % Event count e1 (e2) is the event count that the printer
       %   will signal normal (abnormal) completion on.

       % get the length of c
       length := vector$status(c)
       j := 1
  loop:if j > length then return

       % Now print the j'th character of c
       io$set_status_0011(device, 1, char_to_bits(c(j)), 3, 8)
       io$set_status_1111(device, 1, true, 2, 1)

       % Now wait for the printer to finish and
       %    check to see if an error occurred
       if 2 = ec$await(e1, j, e2, 1) then signal error
       j := j+1
       go to loop
       end unbuffered
```

Figure 5.14. The auxiliary procedure unbuffered.

There are three points to note in this example. First, event counts and sequencers meditate concurrent access to the printer in a simple fashion. Admittedly this problem requires little in the way of synchronization, but the availability of event counts and sequencers to solve the problem at hand is indicative of the utility of AESOP. Second, note the interface between print and the actual device. The device register model permits the lpt cluster to communicate with the device in a simple manner. There are no interrupts from the printer to signal events; instead, the process waits for an indication from the device, via event counts, when the process is ready to look at the results of operations (the Venus operating system[29] works in a similar way using semaphores). This synchronous nature of processes leads to more easily understood programs in a parallel application such as this line printer driver.

Now suppose that a process wants to print vector c in parallel with further computation. Furthermore, suppose that c should not be modified by the lpt cluster. The solution is to spawn a second process to do the printing as shown in Figure 5.15. This permits the executing program to continue execution while the vector is being printed. Moreover, the process pr may only reference c (through the object viewer c'); it may not modify it in any way.

```
% There are 7 operations on vectors. Assume that vector$get
%    is controlled by entry one of access restriction
%    vectors in object viewers.

% Create a read-only view of c, call it c′
ar := vector$create(7, boolean, true)
ar(1) := false
c′ := ov$restrict(c, ar)

% Create a process to do the printing.  Let it run in
%     the default storage area for the current process
%     and use the current GNS as its GNS.
pr := process$create(temp_proc, nil, process$get_default_area(), c′)

% Start the process with default priority and no CPU
%     time limit imposed upon it.
process$start(pr)


% where temp_proc is the procedure:
temp_proc = proc(c:vector[char])
         lpt$print(c)
         end temp_proc
```

Figure 5.15. Using Processes and Access Restriction.

5.6 Conclusions

This chapter has examined several ways to use AESOP. The first four sections examined AESOP from the point of view of languages. Section one showed how to use AESOP′s type mechanism to build objects of extended type. It also showed how parameterized definitions, in particular definitions parameterized by a type, can be achieved. Section two presented four examples of how AESOP′s flexibility in naming and execution, especially the closure facility, can be used in language applications. The use of object viewers for indirection also was presented here. Section three showed how to implement exception handling using AESOP. The termination model was seen to be easily implemented while the implementation of the continuation model had complexity comparable to that of an implementation on a conventional architectures. Section four examined two ways in which AESOP is deficient in handling classical languages such as Algol: unneeded garbage collection and the difficulty of doing call-by-reference parameters and the restrictions imposed by AESOP's local scoping of variables. Finally, section five presented a line printer driver to demonstrate some of AESOP's operating system features.

Chapter Six

Conclusions

This thesis has explored the design of a high level architecture for a personal computer. This chapter examines AESOP and how it has met the goals outlined in chapter one. Some areas for further investigation are also presented.

The primary goal for AESOP was to separate implementation issues from high level language issues. A high level architecture such as AESOP has two demands imposed upon it, both attempting to distort the architecture from the theoretical ideal. Many languages need special features placed in the architecture to accomodate a particular language construct. These influences must be resisted since they conflict with the goal of language independence. On the other hand, many implementations of the architecture need features that allow that implementation to be efficient. These influences must be resisted since they detract from the implementation independence of the architecture. One of the most important results of this thesis concerns the extent to which the design of AESOP has been affected by both of these influences and the subsequent lessons for the designers of other high level architectures.

The influence of languages has been the easiest to resist. The vast majority of languages have the same semantic base. As a result, by providing this semantic base, AESOP is able to support many languages. For instance, capability based naming seems to be the most general of the various naming environments provided by languages. The heap based storage of AESOP seems to be more general than other possibilities (i.e. stack or static storage). The most significant decision that has made AESOP language independent is the decision to treat everything in AESOP as an object. This permits the user to manipulate all aspects of the programming environment. In particular, code and name spaces are full fledged objects in AESOP to allow many language features to be implemented in a simple manner (e.g. CLU iterators, Algol call-by-name). However, the object model is not a panacea. It is a feature of recent languages, so that modeling AESOP after it creates problems in supporting some older languages, as was seen in chapter five.

Resistance to implementation specific features in AESOP has proven to be more difficult. The correctness of an implementation is not at issue. Rather, to be efficient, most implementations need architectural features that collect information from the user. Such information is needed primarily to deal with the physical limitations of the hardware underlying the implementation. It is also needed to suggest optimizations to the implementation. For instance, information about the expected locality of reference aids an implementation based upon a multi-level memory with a slow secondary memory in migrating information within that memory system efficiently. The ability to create vectors of only booleans or of only characters allows an implementation to optimize the storage used for those vectors.

Implementations can acquire the information needed for an efficient implementation in two ways. First, there may be explicit operations in the architecture to aid efficient implementation. For instance, the storage_area$close operation of AESOP exists to provide information about the expected locality of reference in the object memory. Second, an implementation may attach additional semantics to built-in features (including built-in operations). For instance, storage areas are nothing more than quota pools in AESOP. However, the implementation in this thesis assumes that the objects that draw quota from a particular storage area exhibit locality of reference, so that information can be migrated between primary and secondary memory efficiently.

Both techniques detract from implementation independence. The first clutters the architecture with features that are, in some sense, not relevant to solving the task that the user of the architecture has before him. The second approach may result in the user writing programs in a style that is attuned for efficient performance on a particular implementation. Although those programs will be correct on other implementations of the architecture, they may not be as efficient as possible when executed there. Moreover, if the implementation adds very unusual semantics to a built-in feature and the program has been optimized to accomodate those semantics, the program may be hopelessly inefficient on other implementations.

However, some information from the user is essential for efficient implementation. The goal is to acquire it in a way that minimizes the impact of these problems. This is done by making the information sources apply to as many implementations as possible (i.e. to the normal or average implementation). That is, the explicit operations should supply information that most implementations are likely to find useful. The additional semantics that an implementation attaches to architectural features should be chosen to be similar to those that the average implementation adds. In this way, programs written assuming these information sources will run reasonably well on most implementations although their performance may not be as good on the rare or unusual implementation. AESOP's efficiency oriented features meet this goal. Storage areas and the "close" relation on them deal with the multi-level memories that will underly most AESOP implementations. Boolean and character only vectors allow most implementations to perform an important space optimization.

The high level goal of implementation independence was a unifying way of looking at two other views of AESOP: as an actual piece of hardware and as a compiler intermediate language. The next few paragraphs will examine how AESOP has met these goals.

Viewing AESOP as a piece of hardware is associated with two other goals: AESOP should be economically suitable for a personal computer and its efficiency should be comparable to that of a conventional architecture when performing similar tasks. The implementation presented in chapters three and four should meet these goals. If one imagines building AESOP on a microcoded processor, the implementation consists of a large amount of code to perform the various functions outlined in chapters three and four. Some cases are fairly easy (e.g. allocation of storage within a storage area, process management, vectors) while others are more difficult (e.g. physical memory management, garbage collection), but in all cases chapters three and four have described a method of implementation. Since only code is involved, this implies that AESOP becomes more economical as a personal computer as the size of microstores increases. However, as pointed out at the end of chapter four, there are a few places where special hardware (in particular associative memories) will significantly aid the efficiency of an AESOP implementation. These assists seem to be well within the ability of projected and probably current technology to supply cheaply. As a result of these considerations, it is reasonable to expect that AESOP will be economically viable to build as a personal computer at some point in the future. However, its efficiency relative to conventional architectures must be examined more closely.

The implementation presented in chapters three and four should give reasonable efficiency compared to conventional architectures if programs performing similar tasks are compared. AESOP provides a programming environment that permits, as seen in chapter five, many language features to be supported in a straightforward manner. Many of AESOP's features that permit this are gained at no cost compared to conventional architectures. A few of AESOP's features provide facilities not found in conventional architectures (e.g. protection oriented features) but they are provided at some implementation cost. The net effect is that AESOP's performance should be comparable to conventional architectures but with a richer set of facilities. To see this, the time and space efficiency of the implementation of AESOP will be considered.

First, consider time efficiency. A large number of points of comparison are possible cf which only a few of the more prominent are mentioned here. Variables are accessed on AESOP with a single memory reference (by using an offset obtained from an instruction to index off of a register pointing to a name space) just as on a conventional architecture. If the value of a variable is a basic computational type (e.g. booleans, characters, integers), retrieving the variable retrieves its value in both AESOP and a conventional architecture. Accessing a vector element in both cases involves retrieving a pointer to the vector and then performing an indexed access. In both cases, invoking a closure involves creating a small activation record, setting environment pointers and branching. Processes on AESOP are likely to be cheaper than on conventional architectures since AESOP processes embody little state and are easy to schedule. The translation from page identifier to primary memory address that the AESOP implementation performs is similar to what a conventional processor with virtual memory does. Finally, the implementation has been designed assuming that the use of AESOP will follow the subsystem model and that the objects in a storage area will exhibit locality of reference. Thus, to realize its potential efficiency, the user must use AESOP in a way that makes these assumptions true.

There are a few ways in which AESOP may be less time efficient than conventional architectures. In all cases, these inefficiencies come from features that provide important functionality not found on conventional architectures. AESOP's object viewers cost extra memory references to refer through as opposed to directly referring to an object. Conventional architectures avoid this by not providing the fine grained access controls and type extension facilities that object viewers provide. AESOP procedures are somewhat expensive to call because the template LNS must be copied on each invocation. Conventional architectures avoid this by ignoring the problem of initializing a procedure's environment. If AESOP pointers are copied from one storage area to another, cables/links must be searched for and created if not found. While an area is being garbage collected, on average half of all the references to objects in that area will require an indirection through a forwarding pointer to TO space. Finally, the AESOP garbage collector is likely to be more expensive than the garbage collector on conventional systems due to the need to deal with a very large, long term storage system instead of a small, isolated temporary storage system. Thus, in some ways, AESOP may be less time efficient than conventional architectures although this is offset by the gain of some important functionality.

Space efficiency is easier to deal with. The AESOP implementation stores the value of computational variables in the variable itself as on conventional systems. Vectors can be stored with equal efficiency in both cases. This is especially important for the cases of boolean and character vectors. There is no reason to believe that the storage for the remaining types should significantly differ in the two cases since the implementation of AESOP can use the same techniques to represent them as on conventional architectures. In general, AESOP may make less effective use of both logical storage (i.e. storage within a storage area) and physical storage (i.e. the disk storage used to hold storage areas) than programs on conventional architectures since an AESOP implementation embodies general purpose storage management algorithms and not the special purpose algorithms a particular application on a conventional architecture might use.

The key question is how the space/time efficiencies/inefficiencies of AESOP balance out in relation to conventional architectures. This writer believes that, on balance, an AESOP processor should be able to compete with conventional architectures on similar tasks. The value of AESOP's facilities (e.g. support for multiple languages and for the execution of untrusted programs) makes its efficiency acceptable. This is, however, just an opinion and will remain so until AESOP is actually built. The reader must make his own judgement.

AESOP has less successfully met the goal of being a compiler intermediate language. This goal is related to the goal of supporting multiple languages. AESOP does so by providing three features: the correct basic semantics, run-time checking of access to built-in types and object viewers to permit a language system to protect its objects. Chapter five has shown a number of examples of how these features might be used. Supporting multiple languages in the way AESOP does has two advantages over the alternative of having one virtual machine per language system. First, languages can share all of AESOP's built-in types. Moreover, languages can, by mutual agreement, share extended types. Second, languages can share all of AESOP's memory. Thus arbitrarily large data structures are trivially accessible from multiple languages. In addition, shared type managers mean that this data can be of arbitrary type for maximum flexibility.

Efficiency, however, may be a problem. Even though the comments on AESOP's general efficiency mentioned above are true here, the hardware assists mentioned at the end of chapter four are essential to achieving this performance. If the hardware and/or operating system underlying the implementation of AESOP does not provide such assists, programs may run inefficiently. In particular, run-time type checking for the built-in operations, especially on computational data types (e.g. integers), will cost a great deal. Object viewers impose the overhead of additional memory references. Finally, user programs must perform run-time type checking if they are to exist in a world of multiple languages.

An overriding goal for AESOP was that of completeness. The user of AESOP must be able to deal, within AESOP, with all of the realities of computer systems and be able to deal with the many problems that arise in executing programs. In particular, the user must be able to deal with finite resources, buggy or untrusted programs and crashes of the system. The various features normally provided by operating systems must be available or programmable. Also, the human user must be able to interface to AESOP in a way that permits using all of its facilities easily. The next few paragraphs will explore how well this goal has been met.

Finite resources come in three general forms on AESOP: processor, storage and other. AESOP provides a number of controls over processes: starting/stopping, priority and time limits. Storage areas provide a means for controlling the use of finite memory resources by providing a quota mechanism. Other resources, such as I/O devices, must be controlled by user written programs that take advantage of AESOP's treatment of everything as an object. By proper use of these facilities the user is able to control the use of finite resources so as to meet his needs.

Dealing with untrusted programs is just a special case use of the facilities provided for multiple languages and for controlling the use of finite resources. An untrusted program must access only a limited set of objects in prescribed ways. Object viewers and capability-like naming allow this to be enforced. The global name space permits encapsulating such programs so they can have program controlled and monitored access to objects. The resources consumed by an untrusted program must be controlled so that it can not consume resources to the point where the execution of other programs is prevented or adversely affected. AESOP's resource control mechanisms are adequate to prevent this.

AESOP includes various features normally found in operating systems. This is an important step in architectural design as it simplifies the user's view of the world. AESOP provides processes, storage areas, permanent storage, interprocess synchronization/communication and protection. The effect of their inclusion is to eliminate the need for most, if not all, of the operating system normally associated with a computer system. This results in a system that has only two major components (AESOP and the language run time system) rather than the three components normally seen (the hardware, the operating system and the language run time system). The resulting simplicity of the programming interface is significant.

Thus AESOP is able to deal with three important aspects of completeness: finite resources, untrusted programs and operating system features. However, this thesis has left the problems of backup/recovery, system "crashes" (and subsequent recovery) and reliable programs to the implementation and to future research. These are important parts of completeness and their proper solution essential to providing reliable AESOP systems. They have been ignored in this thesis as they are difficult research problems in and of themselves.

The last aspect of completeness is the software that must be written for AESOP to allow human users to effectively interface to AESOP. A command language system must be built to permit the human user to control his programs, create new programs and control the use of AESOP resources. This interface should be designed to give the user easy access to all of the features AESOP provides. The design and construction of this interface represents another interesting research problem. Various auxiliary software such as compilers, linkers and a file system catalog must be written to create a more easily used environment than the basic AESOP environment. These two issues relate to system completeness, how to make AESOP usable, and not to architectural completeness, As such, they are just a side issue to this thesis, albeit an important one.

6.1 Directions for Future Research

This thesis has been a paper design - no implementation of AESOP or of a large application using AESOP has been attempted. No paper design can be completely convincing. As a result there are two major tasks that need to be done initially. First, a language using AESOP as a base should be implemented. Such an endeavor would result in greater confidence that AESOP has the features that are needed in high level applications. A good choice for such an application would be the construction of a cross-compiler to AESOP (e.g. adapt a CLU compiler to produce AESOP code and write the corresponding run-time support that would be needed). Second, AESOP itself should be implemented. It is only in doing an actual implementation that complete confidence will be gained that all of the issues in an AESOP implementation have been addressed by this thesis. These two activities are essential to verifying the ideas of the thesis.

This thesis has brought up a number of related areas that require further research. Perhaps the hardest and most interesting area is that of reliability. AESOP has not provided any architectural features to support reliable programs. Instead, the implementation has been allowed to handle reliability. It is important to understand what features are required in a high level architecture such as AESOP to allow the user to construct his own reliable programs. This is a difficult task if the goal of implementation independence is to be maintained. The complexity and economic constraints of the personal computing environment further complicate the issue.

Beyond this, there is the more basic question of the construction of a reliable implementaτιon of a capability bsed system as AESOP. The problem concerns data that is damaged or lost (e.g. as in a crash of the system). If one or more of the capabilities should be damaged, the result may be chaos - objects that really should be accessible will no longer be and some pointers will be changed to point to random places in memory (possibly to other objects and possibly to random places in memory that may result in failure of the implementation). This is a serious problem whose solution in the cost and complexity constrained environment of a personal computer seems to be difficult.

A related issue to these two is backup and recovery. How should it work in an object based environment? What does it mean to backup the system? Is it a snapshot of the system or does one dump individual objects and somehow remember the relationship between them? How do implementation provided backup/recovery mechanisms and architecturally supplied reliability mechanisms interact? Are they, in fact, incompatible so that only one of them can appear in a given system?

Another research area is the garbage collector proposed in this thesis. When should the garbage collection algorithms be invoked by the system? Are the algorithms suggested in chapter three adequate? If so, how should they be parameterized and what needs to be measured to impose them? If not, what algorithms should be used and is their complexity justified by the increased efficiency of the garbage collection process?

There are questions concerning alternative implementations of AESOP. In many applications, the use of objects is likely to follow a stack oriented discipline. Can an implementation be optimized around that pattern of use in a profitable manner? Can the assumed locality of reference within a storage area be used to compact the format of object references by using so called "short pointer" techniques? In the longer run, implementations of AESOP based upon a greater use of microprocessors to provide additional loci of control for processes and to provide processing power in traditionally passive devices (e.g. primary memory and disk storage devices) need to be explored for their potential exploitation of parallelism both within the implementation and between processes running on the implementation.

A command processor must be built for use on AESOP. What features must it have to permit easy but complete access to all of AESOP's features? How are programs debugged? What should the file system look like? How are the many processes AESOP permits controlled?

This thesis has proposed a particular architectural model of memory. Alternative models exist such as a classical primary/secondary memory of bits or an unstructured object heap as in Snyder's thesis. What level should the model be at (i.e. how much should/can be hidden from the user)? How do efficiency considerations effect the answer to this question? Do better, alternative models exist that are abstract enough to simplify the user's task while still permitting efficient implementations?

If one considers AESOP and its implementation together, a complete system is seen. The semantics of that system contain implementation independent and implementation dependent parts. How is a high level architecture such as AESOP specified to reflect these differences? How are the performance related features specified? Can their specification be "modularized" so as to not effect the semantics of the rest of the system?

AESOP is an interesting exploration into the area of a complete, high level machine architecture. It is only a first attempt into this area and additional work is needed. The readers of this thesis are encouraged to continue exploration into high level architectures that take into account the many issues that a complete computer system <u>must</u> address.

Appendix A

The Operations of the Basic Types

This appendix describes the operations provided by AESOP's built-in type managers. In keeping with the warnings of chapter two, these must be regarded as only suggestions. The reader should feel free to substitute types and operations that suit his particular needs, such substitution being done in the "spirit" of chapter two.

The description of the built-in type managers consists only of the operations provided by those type managers; chapter two should be consulted for a description of the type. The description of an operation begins with a syntactic description of the form:

```
T$op (arg(1), arg(2), ..., arg(N))
        returns (res(1), res(2), ..., res(M))
        signals (sig(1), sig(2), ..., sig(S))
```

This describes the operation named "op" provided by the type manager T. The parameters accepted by the procedure are described by each of the arg(i) and the results provided by the procedure are described by the res(i). Any exceptions raised by this operation are described by the sig(i). A parameter (or result) is described in the form

```
name:type
```

that assigns a name to the parameter for the purposes of the discussion that follows and states that it is of the specified type. One caveat, some operations allow, as an option, a parameter to be nil to signify a special action. An exception is described by the form name(<results>) that states that the exception has the given name and returns the specified results.[1] Following this initial line is a description of what the operation expects in the way of parameters, how it interprets them and what the operation does.

Signal names are unique among all operations. If an exception is raised by more than one operation then it has the same meaning in all operations and is only described at its initial appearance. All operations raise the following exceptions:

invalid_operand_type (i(1), ..., i(N))

too_many_arguments ()

too_few_arguments ()

unexpected_nil_operand (i(1), ..., i(N))

unexpected_deleted_operand (i(1), ..., i(N))

insufficient_access (i(1), ..., i(N))

insufficient_storage

and so they will not be mentioned in the individual descriptions of the operations. The invalid_operand_type exception indicates that the specified arguments to this operation have an incorrect type. The exceptions too_many_arguments and too_few_arguments indicate that the number of arguments passed to this procedure was not within the limits

[1] If more than one result is returned, they are packaged into a vector and that vector is returned due to the restriction imposed by the exception handling mechanism of only one parameter to an exception handler.

as specified by the procedure.[1] The exception unexpected_nil_operand indicates that the specified arguments to this procedure were unacceptable since they referenced a nil object. The exception unexpected_deleted_operand indicates that the specified arguments were deleted objects and as such were unacceptable to this procedure. The exception insufficient_access indicates that the caller has insufficient access to perform the called operation on the specified arguments. The condition insufficient_storage indicates that this operation attempted to create an object but was unable to do so as there was insufficient storage in the area where the create was attempted. .

In the preceding discussion, and in the following, character string names have been used for operations and for exceptions. A particular implementation will bind these character string names to integers.

Boolean

boolean$abcd(x:boolean, y:boolean) returns(z:boolean)

This is a set of 16 operations parameterized by assigning the values 0 and 1 to a, b, c and d. It assigns a boolean value to z based upon the following table:

[1] This condition is raised by the instruction interpreter of the implementation as part of the calling sequence. As such it will be raised not only for built-in procedures but also for user defined ones.

X \ Y	0	1
0	a	b
1	c	d

Thus boolean$0001 performs an and on x and y.

boolean$is_type(x) returns(b:boolean)

Returns true if and only if x is of type boolean.

Characters

char$equal(c1:char, c2:char) returns(b:boolean)

Returns true if and only if c1 and c2 denote the same character.

char$is_type(x) returns(b:boolean)

Returns true if and only if x is of type character.

char$ord(c:char) returns(z:integer)

Returns the index of the character c in the character set being used by the implementation.

char$ord_inv(z:integer) returns(c:char)

Returns the character whose index in the character set being used by the implementation is z.

Closures

closure$create(cs:code_segment, lns:vector, gns:vector)
returns(cl:closure)

Creates a new closure cl consisting of the three arguments as its components.[1] The arguments lns and gns default, if nil, to the LNS and GNS of the executable unit invoking closure$create.

closure$destroy(cl:closure)

Destroys the closure cl invalidating all outstanding references to it.

closure$is_type(x) returns(b:boolean)

Returns true if and only if x is a closure.

closure$run(cl:closure) returns(res(1), ..., res(M))signals(code_seg_destroyed, ...)

Causes the closure cl to be executed and returns any results that cl returns. The signal code_seg_destroyed is returned if the code segment in cl is destroyed. In addition to raising the exceptions based upon its argument, this operation may also raise any exceptions that cl's execution may terminate with.

Code Segment

code_seg$create(rep) returns(cs:code_seg)

Creates a new code segment, cs, from the representation in rep. See chapter two for a discussion of rep.

code_seg$destroy(cs:code_segment)

Destroys the code segment cs invalidating all references to it. If this code segment is still being executed in some process, the

[1] For this and subsequent create operations an optional final argument may be given that denotes the storage area for the newly created object. If omitted, the default storage area for the current process is used.

operation that caused this code segment to be run will return a code_seg_destroyed exception.

code_seg$is_type(x) returns(b:boolean)

Returns true if and only if x is a code segment.

Event Count

ec$await(e(1):ec, c(1):integer, ..., e(N):ec, c(N):integer) returns(which:integer)

Blocks execution of the current process until one of the event counts e(j) has a value at least as large as the corresponding c(j) in which case j is returned in which.

ec$create() returns(e:ec)

Creates a new event count e.

ec$destroy(e:ec)

Destroys the event count e invalidating all outstanding references to it. Any processes waiting on e will be resumed with the ec$await operation signalling unexpected_deleted_operand.

ec$increment(e:ec, n:integer) signals(not_plus, overflow)

Increments the event count e by n. Signals not_plus if n is non-positive. Overflow is signalled if incrementing e by n would cause e to be larger than the implementation supports.

ec$is_type(x) returns(b:boolean)

Returns true if and only if x is an event count.

ec$read(e:ec) returns(count:integer) signals(overflow)

Returns the value of e in count. Signals overflow if e is at its largest possible value.

Integer

integer$abs(x:integer) returns(z:integer) signals(overflow)

Returns x if x is non-negative otherwise returns -x. Overflow is signalled if x is negative and -x is not representable by this implementation.

integer$divide(x:integer, y:integer) returns(z:integer) signals (zero_divide)

z := x/y. Signals zero_divide if y is zero.

integer$equal(i1:integer, i2:integer) returns(b:boolean)

Returns true if and only if i1 and i2 denote the same integer.

integer$is_type(x) returns(b:boolean)

Returns true if and only if x is of type integer.

integer$less_or_equal(x:integer, y:integer) returns(b:boolean)

Returns true if and only if $x \leq y$.

integer$less_than(x:integer, y:integer) returns(b:boolean)

Returns true if and only if $x < y$.

integer$max(x(1):integer, ..., x(N):integer) returns(z:integer)

Returns the largest value from the set {x(i)} in z.

integer$max_int() returns(z:integer) signals(none_such)

Returns the maximum integer supported by this implementation in z. The exception none_such is raised if the implementation does not impose such a limitation.

integer$min(x(1):integer, ..., x(N):integer) returns(z:integer)

Assigns the smallest value from the set {x(i)} to z.

integer$min_int() returns(z:integer) signals(none_such)

Returns the minimum integer supported by this implementation in z.

integer$minus(x:integer, y:integer) returns(z:integer)
signals(overflow, underflow)

z := x-y. An overflow exception indicates that the result of this operation, mathematically, is larger than this implementation can represent. An underflow exception indicates that the result of this operation, mathematically, is smaller than this implementation can represent.

integer$mod(x:integer, y:integer) returns(z:integer)
signals(non_positive_integer)

z is assigned the remainder of dividing x by y. If y is not strictly greater than zero, the exception non_positive_integer is raised.

integer$plus(x:integer, y:integer) returns(z:integer)
signals(overflow, underflow)

z := x+y.

integer$times(x:integer, y:integer) returns(z:integer)
signals(overflow, underflow)

z := x*y.

Notes:

It is possible that a particular implementation will support arbitrary precision arithmetic. If so, max_int and min_int will raise the none_such conditions. Additionally, all operations that produce an integer result may take an additional, optional argument, a storage area, indicating where the integer may be stored. If no such argument is given, the default storage area for the current process is used.

IO

io$create(addr:integer, spec:vector[integer]) returns(dev:io)

signals(none_such, too_late)

Creates a new io object dev whose address is given by addr. The new device has N registers associated with it where N is the length of spec. The vector spec specifies whether each of these N device registers is a status register, an event count register or a buffer register. The exception none_such is raised if the specified device does not exist. The exception too_late is raised if an io$create operation has already been performed on the specified device.

io$destroy(dev:io)

Destroys the logical device dev, invalidating all outstanding references to it. The physical interface between the device and AESOP is made inactive.

io$is_type(x) returns(b:boolean)

Returns true if and only if x is of type io.

io$read_register(dev:io, i:integer) returns(register)

signals(none_such)

Returns the value of the i'th device register of dev. If i is not the name of a valid device register then the signal none_such is raised.

io$set_register(dev:io, i:integer, new) signals(none_such)

Sets the value of the i'th register of dev to new. Signals invalid_operand_type if new is not of the appropriate type for the i'th register of dev.

io$set_status_abcd(dev:io, i:integer, bv:vector[boolean], offset:integer, length:integer) returns(status:vector[boolean]) signals(none_such)

This is a set of 16 operations parameterized by assigning 0 and 1 to a, b, c and d. For j from 1 to length the offset+j-1'th bit of the i'th register of dev is changed to the value in the following table:

x\y	0	1
0	a	b
1	c	d

where x is the value of the j'th bit of bv and y is the value of the offset+j-1'th entry of the status register.

Null

null$is_type(x) returns(b:boolean)

This operation returns true if and only if X is of type null, i.e. returns true only if x = nil.

Object Viewers

ov$access(o) returns(ar:vector[boolean])

Returns, in ar, the access restriction vector associated with the reference o.

ov$destroy(y:ov)

Destroys the object viewer named by y, invalidating all outstanding references to or through it.

ov$destroy_viewed(y:ov)

Destroys the object viewer that y permits modification to, invalidating all outstanding references to or through that object viewer.

ov$extract(o) returns(sealed_object, ar:vector[boolean], revoker:ov)

This operation unseals o, returning the sealed object in sealed_object, the access_restrictions associated with the reference o in ar and an object viewer that permits modifying the object viewer sealing sealed_object in revoker. If the procedure performing this operation is not part of the type manager returned by ov$type(o) then the insufficient_access(1) exception is raised.

ov$is_type(x) returns(b:boolean)

Returns true if and only if x is an object viewer; that is, if and only if x is a reference that permits modification of some object viewer.

ov$modify(y:ov, object, ar:vector[boolean])

Modifies the object viewer named by the revoker y so that its viewed object is object and the access to the viewed object is ar. Both object and ar default to nil. The value nil is always valid as an object field no matter what the type field in the object viewer that y permits to be modified may be. Signals invalid_operand_type(2) if the type of object is not compatible with the type field in the object viewer that y permits to be modified.

ov$restrict(viewed_object, ar:vector[boolean]) returns(X, Y:ov)

This operation returns a new object, X, that represents a restricted view of the object viewed_object. The restrictions are specified by ar. The returned value Y permits modification of the object viewer that is X. The argument ar defaults to nil (no access restrictions).

ov$same_end_objects(o(1), o(2)) returns(b:boolean)

Returns true if and only if o(1) and o(2) currently provide different views of the same object; that is, it returns true if and only if the chains of object viewers named by o(1) and o(2) eventually converge.

ov$same_names(o(1), o(2)) returns(b:boolean)

Returns true if and only if o(1) and o(2) refer to the same object; that is, it returns true if and only if o(1) and o(2) refer to the same non-object viewer object or refer to the same object viewer (i.e. "pointer" equality).

ov$seal(sealed_object, ar:vector[boolean]) returns(X, Y:ov)

signals(not_a_tm)

This operation creates an object, X, of extended type from the representation object sealed_object. The type manager for X, and thus its type, is the type manager invoking this operation. If not invoked by a type manager, not_a_tm is signalled. The permissible operations on X are specified by ar. If ar is not supplied, or is nil, all operations are permitted on X. The returned value Y is a revoker that permits modification of the object viewer that is X.

ov$type(o) returns(t:tm)

Returns the type of the object referenced by o in t. The reference t does not permit the status or destroy operations on the named type manager.

Procedures

proc$call(p:proc, arg(1), ..., arg(N)) returns(res(1), ..., res(M))
signals(code_seg_destroyed, ...)

This operation invokes the procedure p and passes the arguments {arg(i)} to it. The invoked procedure executes in an environment with the global name space of its caller as the global name space of the called procedure. Any results produced by p are returned in {res(i)}. This operation raises various conditions if the argument p is invalid and will also raise any exceptions that p itself may raise.

proc$call_with_gns(gns:vector, p:proc, arg(1), ..., arg(N))
signals(code_seg_destroyed, ...)

This operation invokes p as for proc$call. In addition, p is executed in an environment where gns is the invoked procedure's global name space.

proc$create(cs:code_seg, t_lns:vector, min_args:integer,

max_args:integer, start:integer) returns(p:proc) signals(bad_spec)

Creates a new procedure with cs as its code segment and t_lns as its template local name space. The procedure requires at least min_args arguments but no more than max_args arguments. If max_args is nil, the procedure imposes no limit on the number of arguments expected. The input arguments will be placed starting at location start in the procedure's local name space at run time. The exception bad_spec is raised if any of the integer parameters are non-positive or if max-args is less then min_args.

proc$destroy(p:proc)

Destroys the procedure p, invalidating all outstanding references to it.

proc$is_type(x) returns(b:boolean)

Returns true if and only if x is a procedure.

proc$status(p:proc) returns(cs:code_segment, t_lns:vector,

min_args:integer, max_args:integer, start:integer)

Returns the current status of the procedure p. The result variables have the same meaning as for proc$create.

Process

process$boolean_branch(b:boolean, if_true:integer, if_false:integer)

signals(bad_spec)

Branches to an offset if_true or if_false locations from the current instruction depending upon b being true or false. The exception bad_spec is raised if the specified instruction does not exist.

process$branch_same_end_objects(o(1), o(2), equal:integer, not_equal:integer)

Branches to offset equal (not_equal) if ov$same_end_objects(o(1), o(2)) returns true (false).

process$branch_same_names(o(1), o(2), equal:integer, not_equal:integer)

Branches to offset equal (not_equal) if ov$same_names(o(1), o(2)) returns true (false).

process$create(p:proc, gns:vector, lca:storage_area, arg(1), ..., arg(n)) returns(pr:process)

Creates a new process pr in the stopped state with priority one and no CPU time limit that executes with lca as its initial storage area and begins execution with the instruction proc$call_with_gns(gns, p, arg(1), ..., arg(n)). If nil, gns defaults to the current GNS and lca to the current default storage area.

process$destroy(pr:process)

Destroys the process pr, invalidating all outstanding references to it.

process$get_default_area(pr:process) returns(s:storage_area)

Returns the current default storage area for process pr. If pr is nil, the current process is assumed.

process$integer_compare(a:integer, b:integer, less:integer, equal:integer)

Branches to offset less (equal) if a is less than (equal to) b.

process$is_type(x) returns(b:boolean)

Returns true if and only if x is a process.

process$max_prior() returns(prior:integer)

Returns the maximum possible priority that a process may be assigned. Processes with maximum priority are guaranteed to get some CPU time by AESOP's round-robin scheduler.

process$multi_way_branch(i:integer, offset(1):integer, ..., offset(N):integer)

Branches to an *instruction that is offset*(i) instructions from the current one.

process$return(res(1), ..., res(R))

Causes the current invocation to return normally with the results {res(i)}. If the invoker expects M results, min(M, R) results are actually returned.

process$schedule(pr:process, prior:integer, limit:integer, event:ec)
signals(priority_too_high)

Sets pr's scheduling priority to prior and its CPU time limit to limit. If pr should consume limit units of CPU time total then it enters the stopped state and event is incremented. If limit is nil, no CPU time limit is imposed and event is ignored. Signals priority_too_high if prior is greater than process$max_priority().

process$set_default_area(s:storage_area, pr:process)

Sets the default storage area for the process pr to s. If pr is nil, it defaults to the current process.

process$signal(signal, operand)

Causes the current invocation to return abnormally with signal as the signal name and operand as the signal operand.

process$start(pr:process)

Causes the process pr to enter the runnable state and resume execution.

process$status(pr:process) returns(s:integer, cpu:integer, prior:integer, ec:vector[event_count], limit:integer, event:ec, other)

Returns the current status of pr as follows: current status (runnable, stopped or blocked on some event(s)) in s, cpu time consumed in cpu and current priority in prior. If status indicates that the process is blocked, then an array, ec, of event counts that this process is blocked on is returned (otherwise nil). Also returned are the CPU time limit in limit and an event count, event, that will be incremented when that time limit is exceeded. If limit is nil, no limit is imposed and event is returned as nil.

process$stop(pr:process)

Causes the process pr to cease execution and enter the stopped state.

process$transfer(offset:integer) signals(bad_spec)

Causes execution of the current code segment to continue at offset instructions from the current one.

Sequencer

sequencer$create() returns(s:sequencer)

Creates a new sequencer s with initial value 0.

sequencer$destroy(s:sequencer)

Destroys the sequencer s invalidating all outstanding references to it.

sequencer$is_type(x) returns(b:boolean)

Returns true if and only if x is a sequencer.

sequencer$take(s:sequencer) returns(i:integer) signals(overflow)

Returns the next value in s′s sequence in i. The signal overflow is raised if s is at its largest possible value.

Storage Area

storage_area$close(s:storage_area, s′:storage_area)

Specifies that storage area s should be regarded as being close to s′.

storage_area$create(size:integer, parent:storage_area) returns(s:storage_area)

Creates a new storage area s by drawing size units of quota from parent.

storage_area$delete_all(s:storage_area)

Deletes all objects in the storage area s invalidating all outstanding references to them.

storage_area$destroy(s:storage_area)

Destroys the storage area s and all objects in it invalidating all references to the storage area and to those objects.

storage_area$is_type(x) returns(b:boolean)

Returns true if and only if x is a storage area.

storage_area$not_close(s:storage_area, s′:storage_area)

Specifies that storage area s should no longer be regarded as being close to s′.

storage_area$size(s:storage-area) returns(size:integer, free:integer)

Returns the current size of s in size and the current amount of free quota in s in free.

Note:

master_alloc(size:integer) returns(s:storage_area) signals(cant)

The procedure master_alloc is a built-in procedure that can create storage areas out of nothing. Signals cant if an area of the specified size can not be created. It is supplied at initialization time to the initial process of AESOP.

Type Manager

tm$create(proc(1), ..., proc(N)) returns(t:tm)

Creates a new type manager with proc(i) as the procedure implementing the i′th operation on the newly created type.

tm$destroy(t:type_manager)

Destroys the type manager t, invalidating all outstanding references to it.

tm$is_type(x) returns(b:boolean)

Returns true if and only if x is a type manager.

tm$status(t:tm, i(1):integer, ..., i(M)) returns(proc(1), ..., proc(M)) signals(bad_spec)

Returns as proc(j) the procedure that implements operation i(j). The exception bad_spec is raised if there is no operation i(j) for some j in [1,M].

Vector

vector$create(size:integer, nature:tm, initial_value) returns(v:vector) signals(bad_size, bad_type)

Creates a new vector with "size" elements in it, all initialized to the value initial_value. The exception bad_size is raised if size is non-positive or if size specifies a vector too large for this implementation of AESOP. If nature is nil then v may contain objects of any type otherwise they must be objects provided by the type manager nature. (The parameter nature sets a property of v known as its nature). The exception bad_type is raised if initial_value is not storable in v (i.e. if nature is not nil and if the type of initial_value is not nature).

vector$destroy(v:vector)

Destroys the vector v, invalidating all references to it.

vector$is_type(x, specific:boolean, nature:tm) returns(b:boolean)

If specific is false then nature is ignored and true is returned if and only if x is a vector. If specific is true then true is returned if and only if x is a vector with the same nature as specified by nature.

vector$new_status(v:vector, nature:tm, size:integer, initial_value) signals(bad_type, bad_nature)

Modifies v to have the specified nature and size. If the new size of v is smaller than the old size, initial_value is ignored. If an exception is raised, then v is not modified.

vector$ref(v:vector, i:integer) returns(value) signals(bad_index)

Returns the value of the i'th element of v in value. The exception bad_index is raised if i is not in the range [1,size] where size is the current size of v.

vector$status(v:vector) returns(size:integer, nature:tm)

Returns the current size and nature of v in size and nature respectively.

vector$store(v:vector, i:integer, value) signals(bad_index, bad_type)

Stores value as the value of the i'th element of v. Signals bad_type if value's type is incompatible with v's nature.

Appendix B

The Complete Garbage Collection Algorithm

This appendix describes in detail the garbage collection algorithms used in the implementation of AESOP described in chapter three. The garbage collection algorithms used in AESOP have two parts: the garbage collector used by each individual storage area and a second garbage collector that drives the global mark/sweep garbage collection used to reclaim inter-area cycles of garbage.

There are a few pieces of information that the system maintains for the garbage collection algorithms. Associated with every object is a single bit, its mark bit, that is used to mark objects for the global mark/sweep garbage collection. Each storage area has two flags and two addresses associated with it. The ITID flag (for I Think I'm Done) and the rescan flag are used as part of the global mark/sweep garbage collection algorithm. The address TO_space gives the address of TO space for the storage area while TO_limit is the address of the last object reference in TO space that has been scanned. Every inter-area cable has a gc_in_progress flag and a was_gcing flag associated with it. There are two global flags used to indicate the state of the global mark/sweep garbage collection. The global_gc flag is turned on when a global mark/sweep garbage collection is in progress. The global_sweep flag is turned on when the sweep phase of the global mark/sweep garbage collection is in progress.

There are three algorithms below. The procedure global_gc is responsible for performing the global mark/sweep garbage collection algorithm to reclaim the storage being used by inter-area cycles of garbage. The procedure gc_area performs a garbage collection on a given storage area. The procedure copy_object copies an object from FROM space to TO space and handles the various flags associated with the global mark/sweep garbage collection algorithm.

```
global_gc:procedure()
   % This procedure is responsible for performing the global
   % mark/sweep garbage collection.
   forall S:storage_area
      do S.ITID := true
         S.rescan := false
         do forall O:object in S
            O.mark := false
            end
         end
   % Mark all objects accessible from ROOT
   forall O:object accessible from ROOT
       do Let S be the area that O is in.
          O.mark := true
          S.ITID := false
          end
   global_gc := true
   % Wait for the mark phase to end
   while S.ITID = false for some area S
      do skip end
   % Now do the sweep phase
   global_sweep := true
   forall S:storage_area
       do forall O:object in S
              do if O.mark = false
                    then Return the storage occupied by O to the free
                         pool.
                    else O.mark := false
                 end
       end
   global_sweep, global_gc := false
   end global_gc
```

All of the complexity of this algorithm is hidden in the mark phase which requires a wait for all areas to come to an agreement that the mark phase has completed - this complexity appears below in the description of the gc_area procedure.

```
gc_area:procedure(S:storage_area)
   % Garbage collects area S
   if TO space does not exist
      then Create To space and set TO_space to its address.
      else Set TO_space to the address of TO space.
   TO_limit := TO_space-1
   S.ITID := S.ITID and (not S.rescan)
   forall c:inter-area-cable to S
       do c.gc_in_progress := true end
   forall c:inter-area-cable from S
       c.was_gcing := c.gc_in_progress
   forall O:object accessible from an incoming link to S
       do Let O′ be the inter-area link referencing O.
          call copy_object(O, O′)
       end
   while c.gc_in_progress = true for some incoming cable c to S
        and an object reference exists in TO space above TO_limit
        do Let TO_limit be incremented to the location of the next
           object reference, P, beyond TO_limit in TO space.
           Let P be part of object O′.
           if global_sweep = true and O′.mark = false
              then skip
              else Let O be the object referenced by P.
                   call copy_object(O, O′)
   S.ITID := not S.rescan
   forall c:inter-area-cable from S
      do c.gc_in_progress :=  c.gc_in_progress and not c.was_gcing
      end
   end gc_area

copy_object:procedure(O:object, O′:object)
   % copies the object O, referenced by O′, to TO space
```

```
Let S be the area O is in.
Let S′ be the area that O′ resides in.
Let P be the reference in O′ to O.
Let scanned(X) stand for the following predicate:
     Let S be the area that object X is in.
     Then scanned(X) is equivalent to: S.TO_space ≤ address(X) ≤
        S.TO_limit
     In other words, scanned(X) is true if and only if X has been
     moved to TO space and any object references in X have been
     scanned.
if O is a deleted object
   then Set P to be a reference to a deleted object.
        return
if S = S′
   then if scanned(O)
           then if O′.mark = true and O.mark = false and global_gc =
              true
                 then S.rescan := true
                      return
           else %have not yet scanned O
                if O′.mark = false and global_sweep = true
                   then return
                Copy O to TO space if not already there, leaving a
                   forwarding pointer behind.
                Update P to refer to the copy in TO space.
                O.mark := (O.mark or O′.mark) and global_gc
```

```
                   return
   % Otherwise S and S′ are different areas
   if S is being garbage collected
      then if O is not in TO space
              then copy O to TO space, leaving a forwarding pointer
                   behind.
                   Update P to refer to the copy.
                   O.mark := (O.mark or O′.mark) and global_gc
              else Update P to refer to the copy in TO space.
                   if O.mark = false and O′.mark = true
                      then atomically do O.mark := true
                           if scanned(O)
                              then S.rescan := true
                              e = true
              then atomically do S′.ITID := false
                                 S′.rescan := false
                                 O.mark := true
              end atomically do
   return
   end copy_object
```

One final note: a newly created object should always have its mark bit set to global_gc to allow the global mark/sweep garbage collection algorithm to always terminate.

Two questions are of importance: Does this algorithm terminate? And if it terminates, does it terminate correctly? Each of these will be answered in turn, although a formal proof is beyond the scope of this thesis.

Imagine that a global garbage collection is about to be begun. At this point temporarily freeze the system. Every object in the system is now either accessible or is garbage. The above algorithm ensures that all of this garbage is discovered. For the purpose of the global garbage collection any objects created after this point are considered as accessible (this is ensured by the last point above). This algorithm does a traversal, more or less breadth first, of the graph of accessible objects. Whenever an unmarked object in that graph is discovered, it is marked and, if necessary, the area containing that object is told (by setting its rescan flag) that an object it has already examined during its current garbage collection needs to be traced from again (i.e. have its mark bit propagated). The algorithm terminates for two reasons. First, the set of objects that need to be marked is finite and fixed in size (since newly created objects are automatically marked and thus do not increase the size of the set of accessible but unmarked objects). Second, every object is only scanned once for marking purposes so that no looping need be worried about. The algorithm terminates correctly since it marks all accessible objects, the rescan flag being crucial in preventing mistakes, by preventing the premature setting of ITID flags, and collects all garbage in the sweep phase.

References

[1] Ambler, A.L., et. al., "Gypsy: A Language for Specification and Implementation of Verifiable Programs", SIGPLAN Notices 12, 3(March 1977), pp. 1-10.

[2] Atkinson, R.R., et. al., "Aspects of Implementing CLU", Proceedings of the ACM National Conference, Washington, D.C., December 1978.

[3] Baker, H.G. Jr., "List Processing in Real Time on a Serial Computer", Communications of the ACM 21, 4 (April 1978), pp. 280-294.

[4] Bhandarkar, D.P. and Juliussen, J.E., "Semiconductor Technology: Trends and Implications", Computer Architecture News 7, 1(August 1978), pp.4-14.

[5] Bishop, P.B., "Computer Systems with a Very Large Address Space and Garbage Collection," M.I.T. Laboratory for Computer Science report TR-178, May 1977.

[6] Dahl, O.J., Dijkstra, E.W. and Hoare, C.A.R., Structured Programming, Academic Press, Inc., New York, New York, 1973.

[7] Dennis, J.B. and Van Horn, E.G., "Programming semantics for multiprogrammed computations," Communications of the ACM 9, 3 (March 1966), pp. 143-155.

[8] Dijkstra, E.W., "A Constructive Approach to the Problem of Program Correctness", BIT, 8 (1968), pp. 174-186.

[9] Dijkstra, E.W., "Go To Statement Considered Harmful", Communications of the ACM 11, 3 (March 1968), pp. 147-148.

[10] Dijkstra, E.W., Cooperating Sequential Processes, In Programming Languages, F. Genuys, Ed., Academic Press, New York, 1968.

[11] Fabry, R.S., "Capability-based addressing," *Communications of the ACM 17*, 7 (July 1974), pp. 403-412.

[12] Goodenough, J.B., "Exception Handling: Issues and a Proposed Notation," *CACM 18*, 12 (December 1975), pp.683-696.

[13] Guarino, L.R., "The Evolution of Abstraction in Programming Languages", Carnegie-Mellon University Department of Computer Science Technical Report 78-120, May 22, 1978.

[14] Hansen, P.B., *Operating System Principles*, Prentice Hall, Inc., Englewood Cliffs, N.J., 1973.

[15] Henderson, D.A., "The Binding Model: A Semantic Base for Modular Programming Systems," MIT-LCS TR-145, February, 1975.

[16] Herlihy, M.P., "Communicating Abstract Values in Messages", SM thesis in the M.I.T. Department of Electrical Engineering and Computer Science, expected date of completion January 1980.

[17] Hewitt, C., "Viewing Control Structures as Patterns of Passing Messages," M.I.T. Artificial Intelligence Laboratory, A.I. Memo #410, December, 1976.

[18] Hewitt, C.E., and Atkinson, R., "Specification and Proof Techniques for Serializers", *IEEE Transactions on Software Engineering SE-5*, 1 (January 1979), pp. 10-23.

[19] Hoare, C.A.R., "Monitors: an operating system structuring concept," *CACM 17*, 5 (October 1974), pp. 549-557.

[20] Hoare, C.A.R., "Communicating Sequential Processes", *CACM 21*, 8(August 1978), pp. 666-777.

[21] Hoch, C.G., "Hardware Support for Modern Software Concepts", The University of Texas at Austin, Institute for Computing Science and Computer Applications Technical Report, August 1978.

[22] -----, "PL/1 Language Manual", Honeywell Information Systems Inc., Order nr. AG94, January, 1974.

[23] Kent, S.T., "Implementing Protected Subsystems in Decentralized Environments", M.I.T. Department of Electrical Engineering and Computer Science, Ph.D. thesis in progress, 1979.

[24] Knuth, D., The Art of Computer Programming, Volume 1, Addison-Wesley Publishing Co., Reading, Mass., 1968.

[25] Lampson, B. and Sturgis, H., "Reflections on an Operating System Design", Communications of the ACM 19, 5 (May 1976), pp. 251-265.

[26] Lampson, B., et. al., "Report on the Programming Language Euclid", SIGPLAN Notices 12, 2 (February 1977).

[27] Levin, R., et. al., "Policy/Mechanism Separation in Hydra", Proceedings of the Fifth Symposium on Operating Systems Principles, ACM Operating Systems Review 9, 5(November 1975), pp. 132-140.

[28] Levin, R., "Program Structures for Exceptional Condition Handling," Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, June 1977.

[29] Liskov, B.H., "The Design of the Venus Operating System", CACM 15, 3 (March 1972), pp. 144-149.

[30] Liskov, B.H., and Snyder, A., "Structured Exception Handling," M.I.T. Laboratory for Computer Science Computation Structures Group Memo 155, December, 1977.

[31] Liskov, B.H., et al., "The CLU Reference Manual," CSG Memo # 161, M.I.T. Laboratory for Computer Science, July, 1978.

[32] McCarthy, J., "Recursive Functions of Symbolic Expressions and Their Computation by Machine", Communications of the ACM 3, 4 (April 1960), pp.184-195.

[33] McCarthy, J., et al., LISP 1.5 Programmer's Manual, 2nd edition, M.I.T. Press, Cambridge, Mass. 1965.

[34] McKeeman, W.M., "Language Directed Computer Design", AFIPS Conference Proceedings, 1967 Fall Joint Computer Conference, pp. 413-417.

[35] McMahan, Larry N., "Language Directed Computer Architecture", PhD Thesis, Rice University Department of Electrical Engineering, 1975.

[36] Mitchell, J.G., Maybury, W. and Sweet, R., "Mesa Language Manual, Version 5.0", Xerox Palo Alto Research Center, report CSL-79-3, April, 1979.

[37] Organick, E.I., The Multics System: An Examination of Its Structure, The MIT Press, Cambridge, Mass., 1972.

[38] -----, Proceedings of a Symposium on High-Level-Language Computer Architectures, in SIGPLAN Notices 8, 11(November 1973).

[39] Randell, B. and Russell, L.J., Algol 60 Implementation, Academic Press, London and New York, 1964.

[40] Redell, D.D., "Naming and Protection in Extendible Operating Systems," M.I.T. LCS Technical Report TR-140, November 1974.

[41] Reed, D.P., "Processor Multiplexing in a Layered Operating System," M.I.T. LCS Technical Report TR-164, June 1976.

[42] Reed, D.P. and Kanodia, R.K., "Synchronization with Eventcounts and Sequencers," CACM 22, 2(February 1979), pp. 115-123.

[43] Ritchie, D.M. and Thompson, K., "The UNIX Time-Sharing System", The Bell System Technical Journal, Volume 57, No. 6, Part 2, July-August 1978, pp. 1931-1946.

[44] Saltzer, J.H., "Traffic Control in a Multiplexed Computer System", MIT Project MAC Technical Report 30, July, 1966.

[45] Schroeder, M.D., "Performance of the GE-645 Associative Memory while Multics is in Operation," ACM SIGOPS Workshop on System Performance Evaluation, Harvard University, (April 5-7, 1971).

[46] Siewiorek, D.P., Thomas, D.E. and Scharfetter, D.L., "The Use of LSI Modules in Computer Structures: Trends and Limitations", Computer, July 1978, pp. 16-25.

[47] Snyder, A., "A Machine Architecture to Support an Object-Oriented Language", M.I.T. Laboratory for Computer Science Technical Report 209, March 1979.

[48] Sollins, K.R., "Copying Complex Structures in a Distributed System", MIT Laboratory for Computer Science Technical Report 129, May 1979.

[49] Svobodova, L., Liskov, B., Clark, D., "Distributed Computer Systems: Structure and Semantics," M.I.T. Laboratory for Computer Science Technical Report TR-215, Massachusetts Institute for Technology, March, 1979.

[50] Walker, R.D.H., "The Structure of a Well Protected Computer," Ph.D. Dissertation, University of Cambridge, England, December, 1973.

[51] Wirth, N., "Program Development by Stepwise Refinement", Communications of the ACM 14, 4 (April 1971), pp. 221-227.

[52] Wulf, W.A., "ALPHARD: Towards a language to support structured programming," Carnegie-Mellon University Dept. of Computer Science, April 1974.

[53] Wulf, W., et. al., "HYDRA: The Kernel of a Multiprocessor Operating System", Communications of the ACM 17, 6 (June 1974), pp. 337-345.

[54] Wulf, W.A., Levin, R. and Pierson, C., "Overview of the Hydra operating system development", Proc. Fifth Symposium on Operating Systems Principles, 131, November 1975.

Biographical Note

Allen W. Luniewski was born on August 5, 1952 in Pittsburgh, Pennsylvania. He grew up in the Pittsburgh area, graduating from Penn Hills Senior High School in June, 1970 where he was a member of the National Honor Society.

In September, 1970 he entered Carnegie-Mellon University as a Mathematics major. He was supported by a Brunswick Foundation Scholarship from the Brunswick Corporation and the George S. Dyke Sr. Scholarship from Mine Safety Appliance Company while there. He graduated in May, 1974 with a B.S degree in Mathematics.

In September, 1974 he entered the Massachusetts Institute of Technology as a graduate student in the Department of Electrical Engineering and Computer Science. During his five years at M.I.T. he was supported as a research assistant in the Computer Systems Research Group of M.I.T.'s Laboratory for Computer Science (formerly Project MAC). In January, 1977 he completed his master's degree under Dr. D. Clark with a thesis entitled "A Simple and Flexible System Initialization Mechanism." In June, 1977 he was awarded the degree of Electrical Engineer after completing additional course work.

Mr. Luniewski is a member of the Association for Computing Machinery and its Operating Systems and Programming Languages special interest groups. He is also a member of the Sigma Xi honorary society.

Mr. Luniewski is currently employed by the Systems Development Department of Xerox Corporation in Palo Alto, California where he is working on operating systems for personal computers.

OFFICIAL DISTRIBUTION LIST

Defense Technical Information Center
Cameron Station
Alexandria, VA 22314
12 copies

Office of Naval Research
Information Systems Program
Code 437
Arlington, VA 22217
2 copies

Office of Naval Research
Branch Office/Boston
Building 114, Section D
666 Summer Street
Boston, MA 02210
1 copy

Office of Naval Research
Branch Office/Chicago
536 South Clark Street
Chicago, IL 60605
1 copy

Office of Naval Research
Branch Office/Pasadena
1030 East Green Street
Pasadena, CA 91106
1 copy

New York Area
715 Broadway - 5th floor
New York, N. Y. 10003
1 copy

Naval Research Laboratory
Technical Information Division
Code 2627
Washington, D. C. 20375
6 copies

Assistant Chief for Technology
Office of Naval Research
Code 200
Arlington, VA 22217
1 copy

Office of Naval Research
Code 455
Arlington, VA 22217
1 copy

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps
(Code RD-1)
Washington, D. C. 20380
1 copy

Office of Naval Research
Code 458
Arlington, VA 22217
1 copy

Naval Ocean Systems Center,Code 91
Headquarters-Computer Sciences &
Simulation Department
San Diego, CA 92152
Mr. Lloyd Z. Maudlin
1 copy

Mr. E. H. Gleissner
Naval Ship Research & Development Center
Computation & Math Department
Bethesda, MD 20084
1 copy

Captain Grace M. Hopper, USNR
NAVDAC-OOH
Department of the Navy
Washington, D. C. 20374

1 copy

Mr. Kin B. Thompson
Technical Director
Information Systems Division
(OP-91T)
Office of Chief of Naval Operations
Washington, D. C. 20350
1 copy